NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

MIGUEL ANTÓNIO SOLDADO LOPES

BSc in Electrical and Computer Engineering

A SIGMA-DELTA MODULATION DAC FOR DRIVING A VCXO IN A PLL APPLICATION

MASTER'S IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon

September 2022

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENT OF ELECTRICAL AND COMPUTER ENGINEERING

MASTER THESIS REPORT

A SIGMA-DELTA MODULATION DAC FOR DRIVING A VCXO IN A PLL APPLICATION

MIGUEL ANTÓNIO SOLDADO LOPES

Master's in electrical and Computer Engineering

Adviser: Dr. João Pedro Abreu de Oliveira
Assistant Professor, NOVA University Lisbon

Examination Committee:

Chair: Dr. Luís Filipe dos Santos Gomes - FCT/UNL

Rapporteurs: Dr. Nuno Filipe Silva Veríssimo Paulino - FCT/UNL

Adviser: Dr. João Pedro Abreu de Oliveira - FCT/UNL

MASTER'S IN ELECTRICAL AND COMPUTER ENGINEERING

NOVA University Lisbon
September 2022

A Sigma-Delta Modulation DAC for Driving a VCXO in a PLL Application

ACKNOWLEDGMENTS

Firstly, I want to thank my colleagues, friends, and family for the support during these five years.

A special thanks to my adviser, Dr. João Pedro Oliveira, for all the guidance and encouragement, and also the patience to listen and discuss my ideas.

Thank you to all the professors of the department of electrical and computer engineering which always had their office doors opened for me when I needed help.

I would also like to thank NOVA School of Science and Technology for being my second home during these past five years. An institution where I learnt so much and grew as a person and as an engineer.

Lastly, I would like to thank Dr. Michael Figueiredo, Analog Designer at Concept Silicon/ Aurasemi for giving me the opportunity to work on this project, and for the time invested in helping and guiding me.

ABSTRACT

A Sigma-Delta Digital to Analog Converter (SDDAC) is a system that converts an input digital signal into an analog one by making use of oversampling and sigma-delta modulation to increase the Signal-to-Quantization-Noise Ratio (SQNR).

This thesis studies architectures of SDDACs to drive a Voltage Controlled Crystal Oscillator (VCXO) in a double Phase-Locked Loop (PLL) system. The goal is to achieve a phase noise due to the SDDAC that is lower than the intrinsic VCXO phase noise. The VCXO has a free-run frequency of 122.88 MHz, and its output is also used for a sampling clock in the digital filter that precedes the SDDAC. This study will allow choosing the best SDDAC that satisfies the noise requirements.

The studies are done by discrete time simulations on a developed tool that consists of high-level models of SDDACs that take into consideration many digital sigma-delta modulator architectures, and a variety of non-idealities and noise on a current steering Digital-to-Analog Converter (DAC) and on a gm-C low-pass reconstruction filter. The developed tool is presented in this thesis, as is a study of the phase noise due to SDDACs is shown as a proof of concept.

This work also includes the design of some of the main analog blocks in a 10-bit 122.88 MHz current steering DAC in 130nm Complementary Metal-Oxide-Semiconductor (CMOS) technology.

Keywords: Sigma-Delta Modulation, Digital to Analog Conversion, Voltage-Controlled Crystal Oscillator, Phase Noise, High-Level Modelling, Current Steering Digital-to-Analog Conversion.

RESUMO

Um conversor digital-analógico sigma-delta (SDDAC) é um sistema que converte um sinal digital de entrada num sinal analógico fazendo uso de sobre amostragem e modulação sigma-delta para aumentar o rácio entre o sinal e o ruído de quantização.

Nesta tese, são estudadas arquiteturas de SDDACs para fazer o controlo de um oscilador de cristal controlado por tensão (VCXO) num duplo "phase-locked loop" (PLL). O objetivo é atingir um ruído de fase devido ao SDDAC que seja inferior ao ruído de fase intrínseco ao VCXO. A frequência de livre funcionamento do VCXO é 122.88 MHz, e a sua saída é utilizada para criar o relógio de amostragem do filtro digital que precede o SDDAC. Este estudo permitirá escolher o melhor SDDAC que satisfaça os requisitos de ruído de fase.

Os estudos são feitos através de simulações em tempo discreto numa ferramenta que foi desenvolvida e que consiste em modelos de alto nível de SDDACs que têm em consideração várias arquiteturas de moduladores sigma-delta digitais, e diversas não-idealidades e ruído em conversores digitais analógicos por desvio de corrente e no filtro de reconstrução passa-baixo gm-C. A ferramenta desenvolvida é apresentada nesta dissertação assim como um estudo do ruido de fase devido a SDDACs que serve como prova de conceito.

Este trabalho também inclui o “design” de alguns dos blocos analógicos mais importantes de um conversores digitais analógicos por desvio de corrente com 10 bits e uma frequência de amostragem de 122.88 MHz.

Palavras-chave: Modulação Sigma-Delta, Conversão Digital-Analógico, Oscilador de Cristal Controlado por Tensão, Ruído de Fase, Modelo de Alto-Nível, Conversão Digital-Analógico através do Desvio de Corrente.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PYTHON CODE

ACRONYMS

ADC	Analog to Digital Converter
CAS	Cascode
CIFB	Cascade of Integrators with Feedback
CIFF	Cascade of Integrators with Feedforward
CN	Common Node
CRFB	Cascade of Resonators with Feedback
CRFF	Cascade of Resonators with Feedforward
CSDAC	Current Steering Digital to Analog Converter
DAC	Digital to Analog Converter
DC	Direct Current
DEM	Dynamic Element Matching
DNL	Differential Non-Linearity
DSDM	Digital Sigma-Delta Modulator
EF	Error Feedback
EFM	Error Feedback Modulator
ENOB	Effective Number of Bits
FFT	Fast Fourier Transform
FPGA	Field-programable Gate Array

GND	Ground
HKEFM	Hybrid-Key Error Feedback Modulator
HPF	High-pass Filter
IC	Integrated Circuits
IBN	In-band Noise
INL	Integral Non-Linearity
KCL	Kirchhoff's Current Law
LPF	Low-pass Filter
LSB	Least Significant Bit
MASH	Multi-Stage Noise Shaping
MOS	Metal-oxide-semiconductor
MOSFET	Metal-oxide-semiconductor Field-effect Transistor
NMOS	N-type Metal-oxide-semiconductor
NTF	Noise Transfer Function
OF	Output Feedback
OFM	Output Feedback Modulator
OSR	Oversampling Ratio
PLL	Phase-locked Loop
PMOS	P-type Metal-oxide-semiconductor
PSD	Power Spectral Density
RC	Resistor-Capacitor
RL	Load Resistor
SDDAC	Sigma-Delta Digital to Analog Converter
SDM	Sigma-Delta Modulator
SERDES	Serializer-Deserializer

SFDR	Spurious-free Dynamic Range
SNDR	Signal-to-Noise-plus-Distortion Ratio
SNR	Signal-to-Noise Ratio
SQNR	Signal-to-Quantization-Noise Ratio
STF	Signal Transfer Function
SW	Switch
TF	Transfer Function
THD	Total Harmonic Distortion
VCAS	Cascode Voltage
VCO	Voltage Controlled Oscillator
VCXO	Voltage Controlled Crystal Oscillator
VDD	Drain-Drain Voltage
VDS	Drain-Source Voltage
VGS	Gate-Source Voltage
VOV	Overdrive Voltage
VSG	Source-Gate Voltage
VTH	Threshold Voltage
XO	Crystal Oscillator

1

INTRODUCTION

The signal conversion from analog to digital and vice-versa is something which every data acquisition system and every device performing computational and signal processing tasks heavily relies on. Nature can be perceived as analog and comprised of continuous signals with which we interact and process in our everyday lives, but with today's modern technology, these analog signals must be processed and generated by computational systems which work with discrete digital data [1]. The dominance of digital integrated circuits (ICs) in computational and signal processing tasks is ever-growing due to the increase in speed of operations and reduction in volume occupied by digital ICs. So, the task of converting signals from one domain to another is extremely important.

Analog to Digital converters (ADC) and Digital to analog converters (DAC) are two of the most important blocks in modern electronics. This work will be more focused on DACs rather than ADCs. These systems take digital data (bits) as input and output analog quantities like voltages or currents. The input bits may come from a microprocessor, an Application-Specific Integrated Circuit (ASIC), a Field-Programmable Gate Array (FPGA), or any other digital electronics device and may be used to perform any kind of control over an analog system, in radio and audio applications, in data acquisition or distribution systems, and in many other areas.

The scope of this work is about one particular application of DACs, which is to perform extremely high accuracy control of a voltage-controlled crystal oscillator (VCXO) [2] used in a phase-locked loop (PLL) application. This type of oscillator produces a periodic signal with some frequency that depends on an input voltage. So, a control digital signal used to tune the oscillating frequency and must be converted to a voltage with as low associated noise as possible. This must be done by a high-resolution DAC. In this particular PLL application the

resolution needed is so high that conventional Nyquist DACs are not capable of this and an oversampling converter with sampling frequency OSR (oversampling ratio) times greater than the Nyquist frequency may be used to convert a very long digital word (48 bits) into a very accurate analog voltage. The most widely used technique for oversampling converters is sigma-delta modulation (SDM) which uses oversampling and quantization error shaping to trade speed for resolution and analog circuit accuracy for digital-circuit complexity.

This thesis presents the study and high-level modelling of a low noise sigma-delta Digital to Analog converter used for driving a VCXO incorporated in a double-loop PLL [11]. This system is supposed to act as a master clock distributor for all the blocks in the system (DAC, ADC, FPGA, SERDES, etc.), so it must exhibit perfect timing and synchronism between all the clocks it generates. That is why there is a need for a double-loop PLL in this application.

This master thesis project was suggested by Aurasemi engineers, had the support of Dr. Michael Figueiredo, Director and Analog Designer at Concept Silicon, and fits in a new project tool to be used by them.

The main goal is to achieve a noise level equal or lower to that of the 122.88 MHz VCXO that is being controlled by a 48-bit digital word signal with a bandwidth from 0.1 Hz to 1 kHz. Two approaches will be studied: a 48-bit to M-bit digital SDM followed by a M-bit DAC to drive the VCXO, or a 48-bit to 1-bit sigma-delta DAC to drive the VCXO.

In this thesis, there is an initial study of sigma-delta modulation, more specifically, digital sigma-delta modulators in section 2, and also the current steering architecture for the digital to analog converter (DAC) in section 3. Section 4 shows the main study regarding the system to use in the PLL, and it is done by developing high-level models that can be simulated in discrete time domain and offer insight on how the noise generated by these will affect the output of the VCXO. The modelling work is complemented by the design of a 122.88 MS/s 10-bit current steering DAC (CSDAC) in a CMOS technology. Design and simulation of current sources, biasing circuit and switch drives are shown in section 5.

2

SIGMA-DELTA MODULATION

Data converters can be classified into two different categories depending on the sampling rate they use. Nyquist-rate converters use sampling rates slightly above the Nyquist frequency established by the Nyquist criterion which says that one only needs a sampling frequency of at least twice the bandwidth of the signal to be able to reconstruct it. Even though these converters can achieve very high speeds and a reasonable resolution, this last characteristic is limited by the matching accuracy of physical components that make the converter. If one wants to achieve a higher resolution with a higher effective number of bits, an oversampling converter may be used. These converters use sampling rates much higher than the Nyquist's. Each output sample is calculated using all preceding input values, meaning that memory elements are used in its structure [3, pp. 1-4].

The scope of this work are oversampling converters, especially Sigma-Delta Digital to Analog Converters which make use of ΣΔ modulation to greatly increase the signal to noise ratio achieved by them, hence increasing the effective resolution of the converter.

2.1 Sampling, Noise and Quantization

As previously mentioned, oversampled converters use sampling frequencies much higher than the Nyquist rates. An evenly sampled signal g(t) with sampling period T can be represented as the impulse train in equation 2.1.

$$g^*(t) = \sum_{k=-\infty}^{+\infty} g(kT) \cdot \delta(t - kT) \tag{2.1}$$

Equation 2.1 is in the time domain and has its equivalent form in the frequency domain represented by its Fourier transform which is shown in equation 2.2.

$$G^*(\omega) = \frac{1}{T}\sum_{n=-\infty}^{+\infty} G(\omega - n\omega_s), \tag{2.2}$$

$$\omega_s = \frac{2\pi}{T}$$

ω_s is the sampling frequency in radians per second and G(ω) is the Fourier transform of g(t) [4]. This means that sampling a signal in time domain maps to a train of spectral replicas centered at multiples of the sampling frequency in the frequency domain. This property is known as imaging and is depicted in figure 1 for some signal with some spectrum representation.

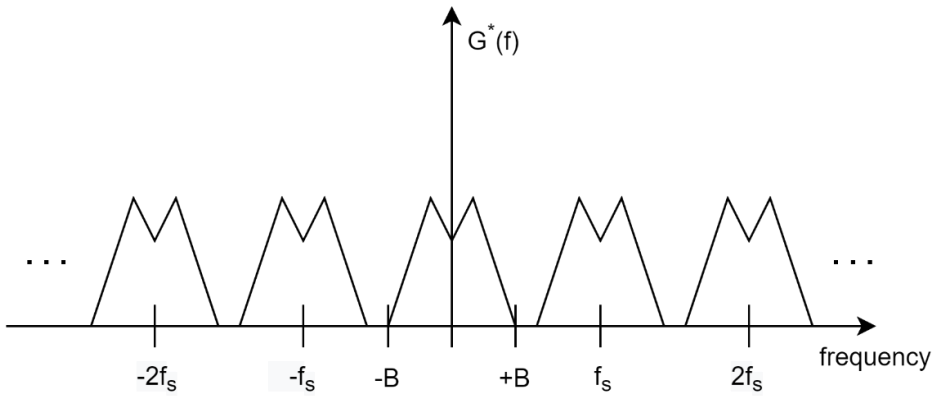


Figure 1 – Imaging in frequency domain.

By increasing the sampling frequency to a much higher value than the Nyquist's, these spectral replicas appear further apart which is beneficial for anti-aliasing performance, increased resolution, and reduced in-band noise (IBN) is reduced.

In signal processing a measurement widely used is the signal to noise ratio, also known as SNR, which is the ratio between the power of the signal and the power of the noise that it contains. It can be calculated by equation 2.3. Its relationship with the effective number of bits (ENOB) is given by equation 2.4.

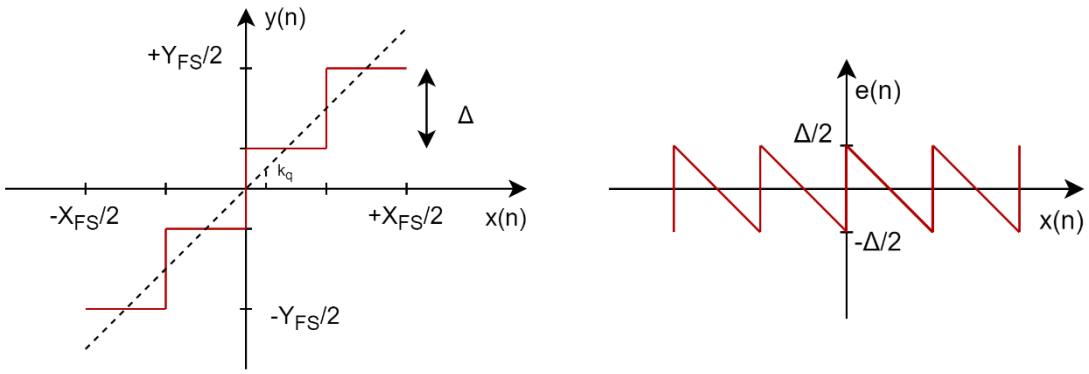
$$SNR = 10 \cdot \log_{10}\left(\frac{P_{signal}}{P_{noise}}\right) \quad [dB] \tag{2.3}$$

$$ENOB = \frac{SNR_{max} - 1.76}{6.02} \tag{2.4}$$

So, by increasing the signal to noise ratio, we are increasing the effective resolution of the converter. The difference between the input value which results in the maximum value of the SNR and the input value which results in a SNR of 0 dB is usually called dynamic range (DR). When referring to SQNR one is referring only to the ratio between the power of the signal and the power of the quantization noise and not taking into consideration any other sources of noise.

Data converters use quantization to convert digital words to analog voltages and vice-versa. Figure 2 shows an example of a 2-bit quantization where Δ is the quantization step and can be calculated as in equation 2.5, where y_{FS} is the output full-scale of the converter and N is the resolution in bits.

$$\Delta = \frac{y_{FS}}{2^N - 1} \tag{2.5}$$



a) b)

Figure 2 – 2-bit quantizer a) Transfer characteristic b) Error characteristic.

This process introduces an error called quantization error which is correlated with the input signal and can be shown to be bounded within [-Δ/2, +Δ/2]. If we consider that there is enough noise in the system, this quantization error turns into quantization noise, completely uncorrelated with the input signal. Under these circumstances the quantization error (e) can be seen as a white noise random process with uniform probability distribution in the range that was mentioned above, and its power spectral density (PSD or S_E) over the range [-fs/2, +fs/2] is equal to equation 2.6 as was demonstrated in [5, pp. 5-8].

$$S_E = \frac{\Delta^2}{12 \cdot fs} \tag{2.6}$$

One important definition is the oversampling ratio (OSR) which is given by equation 2.7.

$$OSR = \frac{fs}{2B} \tag{2.7}$$

If we consider an ideal filter capable of filtering out all the out of band spectrum, the in-band quantization noise (IBN) can be calculated as in equation 2.8.

$$IBN = \int_{-B}^{+B} S_E \cdot df = \frac{2B \cdot \Delta^2}{12 \cdot fs} = \frac{\Delta^2}{12 \cdot OSR} \tag{2.8}$$

So, as was said in the previous section, the greater the OSR, the smaller the IBN. This property is shown in figure 3.

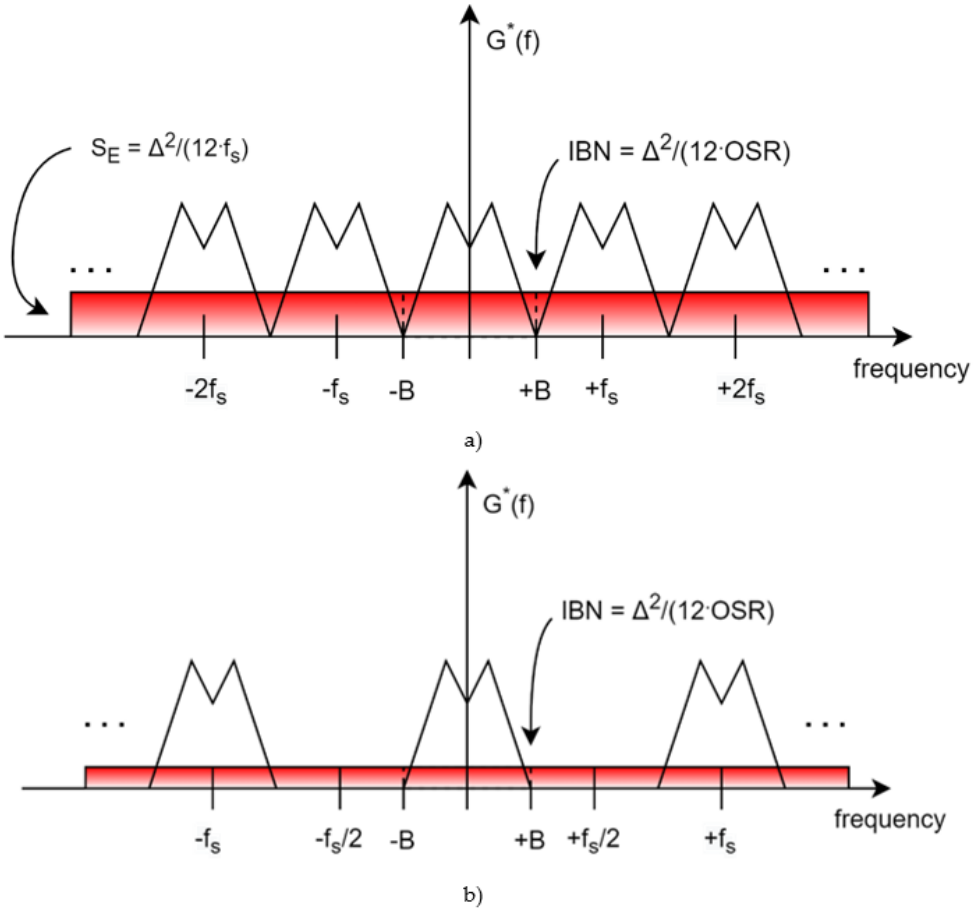


Figure 3 — Aliasing and effect of oversampling in the IBN a) fs = 2B b) fs = OSR·2B.

2.2 Noise Shaping in Sigma-Delta Modulation

The main goal of sigma-delta modulation is to achieve something called noise shaping where the quantization noise, previously considered to behave like white noise, is shaped to high frequencies, away from the signal's bandwidth. This process is illustrated in figure 4.

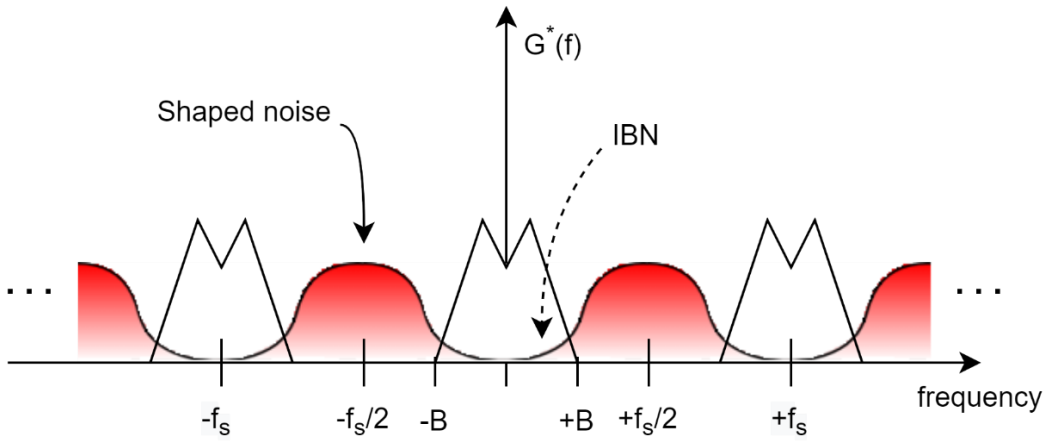


Figure 4 – Noise shaping in frequency domain.

Note that, for this process to work, there is a need for the spectral replicas to be further apart in the frequency domain, and that is where the oversampling mentioned before plays a big role. By oversampling the input signal with a rate much higher than the Nyquist's, we make room for the noise to be shaped to higher frequencies in between the spectral replicas of the sampled signal. One particular easy way to achieve this in low-pass signals is to find a system which processes the noise like a high-pass filter (HPF) and the signal as a low-pass filter (LPF). A noise transfer function (NTF) in the z domain, which processes noise like a HPF is shown in equation 2.9, where L refers to the order of the filter in the modulator.

$$NTF(z) = (1 - z^{-1})^L \qquad (2.9)$$

$$z = e^{s \cdot T_s} = e^{j \cdot 2\pi f / fs}$$

Considering equation 2.8 again, because in the case of the modulator the noise is processed by a noise transfer function the new in-band noise power can be calculated as

$$IBN = \int_{-B}^{+B} S_E \cdot |NTF(f)|^2 \cdot df, \qquad (2.10)$$

and if we consider that fs is much bigger that f, because of the oversampling, the $|NTF(f)|^2$ can be approximated as shown below and the IBN can written as in equation 2.11.

$$|NTF(f)| = \left|1 - e^{j2\pi f/fs}\right|^L \approx \left|1 - \left(1 + j2\pi \frac{f}{fs}\right)\right|^L = \left(\frac{2\pi f}{fs}\right)^L$$

$$IBN \approx \frac{\Delta^2 \cdot \pi^{2L}}{12 \cdot (2L+1) \cdot OSR^{2L+1}} \qquad (2.11)$$

From equation 2.11 it is possible to conclude that with the increase in the order L of the NTF there is a significant reduction in the IBN power.

In the case of sigma-delta DACs, the quantizer performs a truncation which reduces the word length (in bits) presented at the input of the system. By reducing the word length there is a loss of information, our data is described in a less precise way, and quantization noise is generated. It is the modulator's job to shape this noise to higher frequencies away from the signal's bandwidth so that it can be filtered out afterwards. This is done by introducing the opposite error of one cycle in the next cycle. Over time, these errors should average out [3, pp. 34-35].

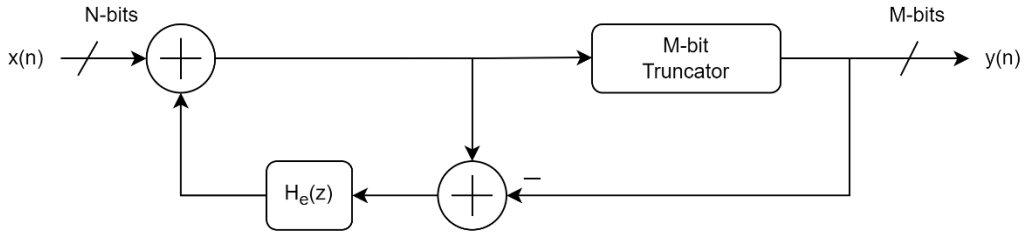
The general configuration of a digital Sigma-Delta modulator used in DACs is shown in figure 5a and is called error-feedback (EF) configuration. The truncation process is highly non-linear so, in order to simplify the analysis of the system, it can be substituted by its linear model as shown in figure 5b. This model can be simplified not to have the transfer function $H_e(z)$ (or NTF(z)) in the feedback path but rather before the truncation as shown in figure 5c. For this, the transformation in equation 2.12 is used.

$$H(z) = \frac{1 - H_e(z)}{H_e(z)} \tag{2.12}$$

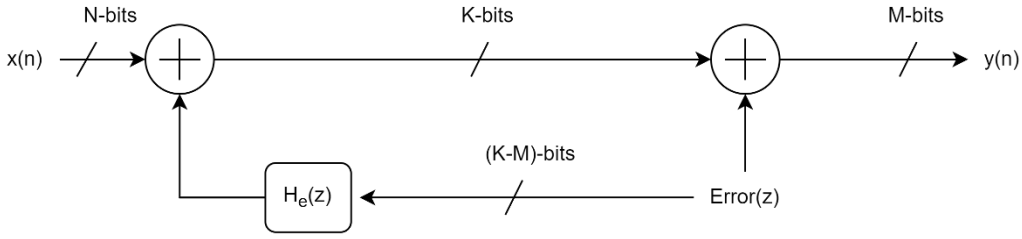
So, the z-domain behavior of the system with respect to the input signal and the error signal is given by the signal transfer function (STF) and the NTF in equations 2.13.

$$STF(z) = \frac{H(z)}{1 + H(z)}, \qquad NTF(z) = \frac{1}{1 + H(z)} \tag{2.13}$$

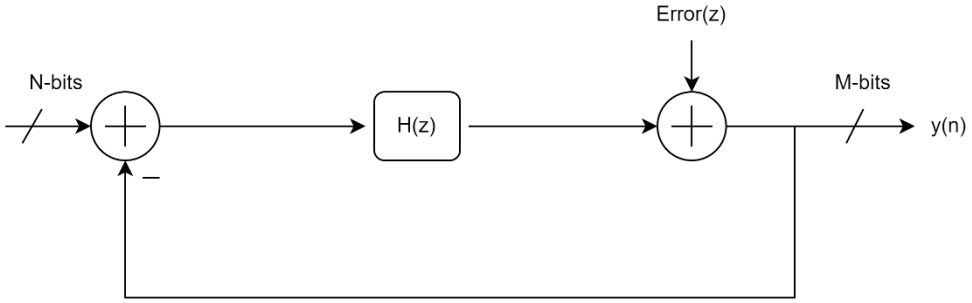
Generally, H(z) is chosen so that the STF is approximately 1 in the signal's bandwidth and the NTF is close to 0 in that same frequency range. This means that the noise is highly reduced around the signal, and the signal is virtually untouched by the modulator.



a)



b)



c)

Figure 5 – a) EF configuration b) output-feedback (OF) linear model c) OF simplified linear model.

2.3 First order Sigma-Delta Modulators

A simple first order Sigma delta modulator can be achieved by setting L equal to 1 in equation 2.9. Using equation 2.12 we get that

$$H(z) = \frac{1 - NTF(z)}{\mathrm{NTF}(z)} = \frac{1 - 1 + z^{-1}}{1 - z^{-1}} = \frac{z^{-1}}{1 - z^{-1}} \tag{2.14}$$

which is the transfer function of a discrete time integrator with a one sample delay. Thus, the first order modulator can be implemented as follows in figure 6.

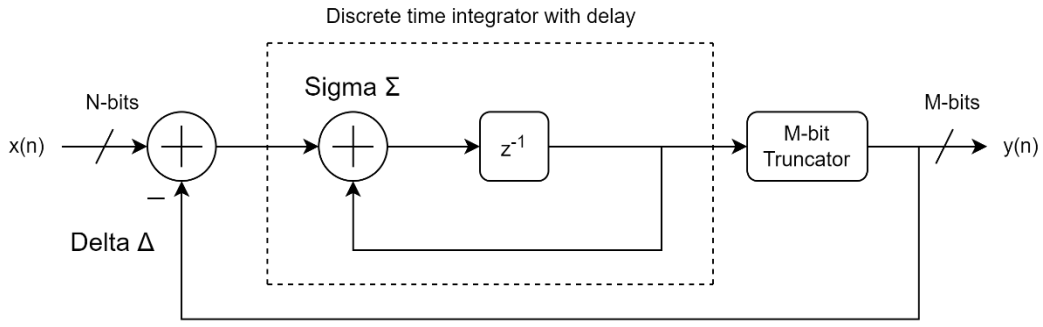


Figure 6 – First order sigma-delta modulator.

Note that, using equation 2.13, the STF(z) of this system will be

$$STF(z) = \frac{H(z)}{1 + \mathrm{H(z)}} = \frac{{z^{-1}}/{(1 - z^{-1})}}{1 + {z^{-1}}/{(1 - z^{-1})}} = \frac{z^{-1}}{1 + z^{-1} - z^{-1}} = z^{-1}, \tag{2.15}$$

which is just a one sample delay on the signal. Using equation 2.11 to determine the IBN of a first order modulator, we see that it achieves the IBN in equation 2.16.

$$IBN_{MOD1} \approx \frac{\Delta^2 \cdot \pi^2}{36 \cdot OSR^3} \tag{2.16}$$

Using a full-scale symmetric input signal with N bits whose power equals to

$$Pin_{FS} = \left(\frac{(2^N - 1) \cdot \Delta}{2\sqrt{2}}\right)^2 \quad [V_{rms}^2], \tag{2.17}$$

yields a maximum SQNR of

$$SQNR_{max_{MOD1}} \approx 10 \log_{10}\left(\frac{9 \cdot (2^N - 1)^2 \cdot OSR^3}{2\pi^2}\right) \quad [dB]. \tag{2.18}$$

If we use a truncation to 1 bit (therefore a simple 1-bit converter) and an oversampling ratio equal to 128, we get a maximum SQNR of approximately 60 dB which is equivalent to approximately a 10-bit Nyquist converter. Also note that if we double the OSR we get an increase of 9 dB in the SQNR.

2.4 Second order Sigma-Delta Modulators

Like shown in the previous section, if one wants do design a simple second order modulator, one can just set L = 2 in equation 2.9 and use it in equation 2.12 to determine the following loop filter transfer function:

$$H(z) = \frac{1 - NTF(z)}{\mathrm{NTF}(z)} = \frac{1 - (1 - z^{-1})^2}{(1 - z^{-1})^2} = \\ = \frac{1 - 1 + 2z^{-1} - z^{-2}}{(1 - z^{-1}) \cdot (1 - z^{-1})} = \frac{z^{-1} \cdot (2 - z^{-1})}{(1 - z^{-1}) \cdot (1 - z^{-1})}. \tag{2.19}$$

It can be shown that such transfer function can be created having a cascade of two first order modulators, as it could be expected. The block diagram of this system is shown in figure 7.

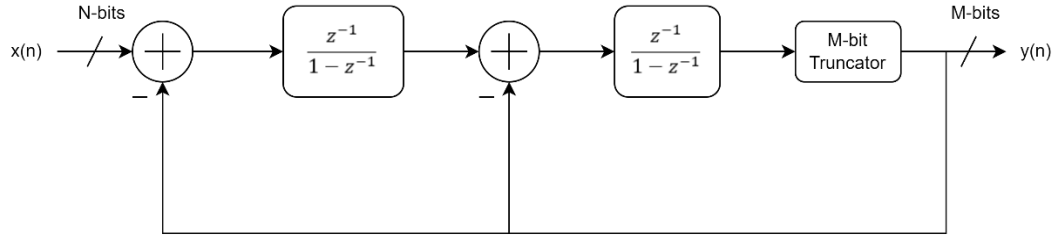


Figure 7 – Pseudo second order digital Sigma-Delta modulator.

But this modulator does not actually perform as we wanted it to in equation 2.13. For that, we must introduce some gain factors in the direct and feedback paths as shown in figure 8. Another way to make it perform like the loop transfer function in 2.13 would be to remove the delays from the integrator blocks and to add a delay in the feedback path, but by adding gain factors as free variables, we increase the number possibilities of NTF and STF that we can construct we just one architecture.

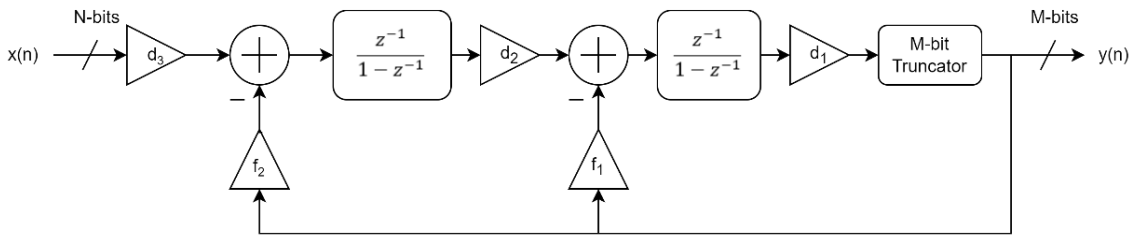


Figure 8 – Second order digital sigma-delta modulator with gain factors.

Analyzing the system in figure 8 we get the equations in 2.20.

$$Y(z) = E(z) + d_1 \cdot \frac{z^{-1}}{1-z^{-1}} \cdot \left(d_2 \cdot \frac{z^{-1}}{1-z^{-1}} \cdot \left(d_3 \cdot X(z) - f_2 \cdot Y(z) \right) - f_1 \cdot Y(z) \right)$$

$$Y(z) = E(z) \cdot NTF(z) + X(z) \cdot STF(z) \qquad with$$

$$NTF(z) = \frac{(1-z^{-1})^2}{1 - z^{-1} \cdot (2 - f_1) + z^{-2} \cdot (d_1 d_2 f_2 - f_1 + 1)} \qquad and \tag{2.20}$$

$$STF(z) = \frac{d_1 d_2 d_3 \cdot z^{-2}}{1 - z^{-1} \cdot (2 - f_1) + z^{-2} \cdot (d_1 d_2 f_2 - f_1 + 1)}$$

In order to reduce the denominators to 1, $d_1 = d_2 = d_3 = 1$, $f_1 = 2$, $f_2 = 1$. And we get

$$STF(z) = z^{-2}, \qquad NTF(z) = (1 - z^{-1})^2. \tag{2.21}$$

Using equation 2.11 to determine the IBN of a second order modulator, we see that it achieves an IBN of

$$IBN_{MOD2} \approx \frac{\Delta^2 \cdot \pi^4}{60 \cdot OSR^5}, \tag{2.22}$$

which, using a full-scale input signal yields a maximum SQNR of

$$SQNR_{max_{MOD2}} \approx 10 \log_{10} \left(\frac{15 \cdot (2^N - 1)^2 \cdot OSR^5}{2\pi^4} \right). \tag{2.23}$$

If we use a truncation to 1 bit (therefore a simple 1-bit converter) and an oversampling ratio equal to 128, we get a maximum SQNR of approximately 94 dB which is equivalent to approximately a 15-bit Nyquist converter. Also note that if we double the OSR we get an increase of 15 dB in the SQNR.

2.5 Sigma-Delta Modulators performance measurement and simulation

In this section the simulation and performance measurement of sigma-delta modulators is explained. These systems tend to be simulated in time domain using its difference equation

which, for example, for a first order digital modulator follows from the system in figure 6 and yields the difference equation in 2.24, where the signal t[n] is the input of the quantizer.

$$y[n] = truncation(t[n])$$
$$t[n] = t[n-1] + x[n-1] - y[n-1]. \tag{2.24}$$

Figure 9 shows the output of a first-order error feedback digital sigma-delta modulator with 1-bit truncation and OSR equal to 128 in time domain when excited with a sine wave. Figure 10 shows the output frequency spectrum of the system.

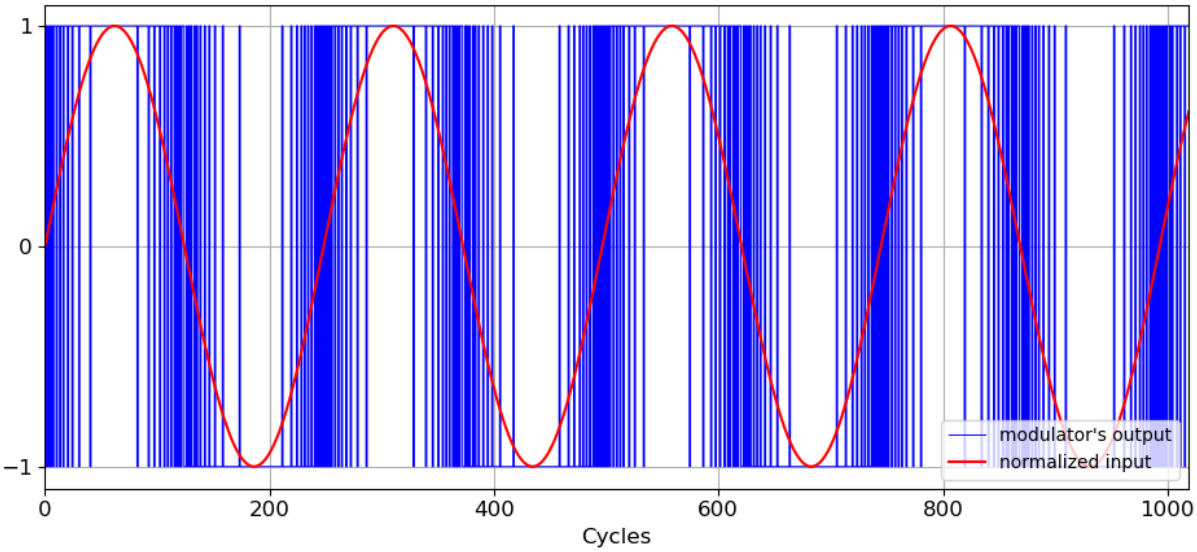


Figure 9 – Input and output of a first order ΣΔ modulator in time domain.

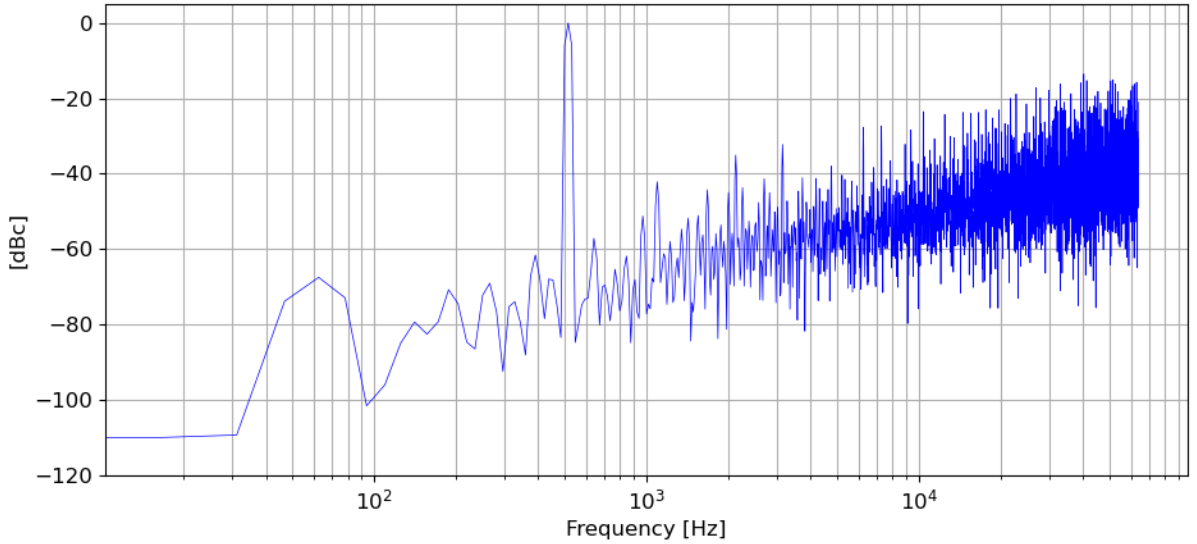


Figure 10 – Output spectrum of a first order ΣΔ modulator.

It can be seen in figure 9 the input and the output of the modulator in the time domain. At first glance there does not seem to be any obvious relationship, but with more attentive inspection we can clearly see that for bigger input values the system outputs a longer stream of +1 and for smaller input values it outputs a longer stream of -1, resulting in less variation in the output. Output values varying rapidly are result of input values closer to 0. This means that to make sense of these plots we need to think about the average output in some period of time and not an individual sample. The information about the input signal is encoded in the output and corrupted by a lot of noise, but because this noise is situated mostly at high frequencies outside the baseband of the signal, as it can be observed in figure 10, it can be filtered out rather easily without changing the input, and an output truncated to 1-bit (in this case) will seem to have a much bigger resolution after the filtering. It is important to note in the output spectrum that noise shaped by the first order modulator is increasing at a rate of 20 dB per decade. A second order system would shape the noise at twice that rate.

Some important performance metrics and definitions that haven't been mentioned are depicted in figure 11. The SFDR is the spurious free dynamic range and can be defined as the ratio between the power of the fundamental signal and the strongest of all the other harmonics in the frequency spectrum as seen in equation 25.

$$\mathrm{SFDR} = 10\log_{10}\frac{P_{in}}{P_{strongestH}} \tag{2.25}$$

The total harmonic distortion (THD) can be defined as sum of the powers of the harmonics in the signal that are not the fundamental.

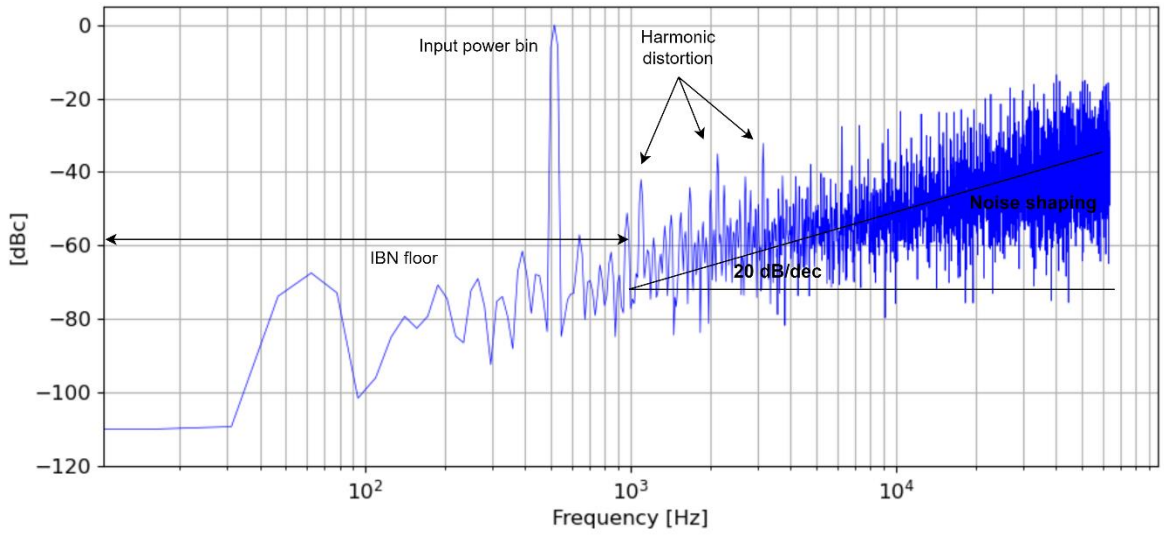


Figure 11 – Some important metrics in the output spectrum.

Another relationship that is important to visualize is the SNR as a function of the input's amplitude. An example for the modulator mentioned previously is depicted in figure 12.

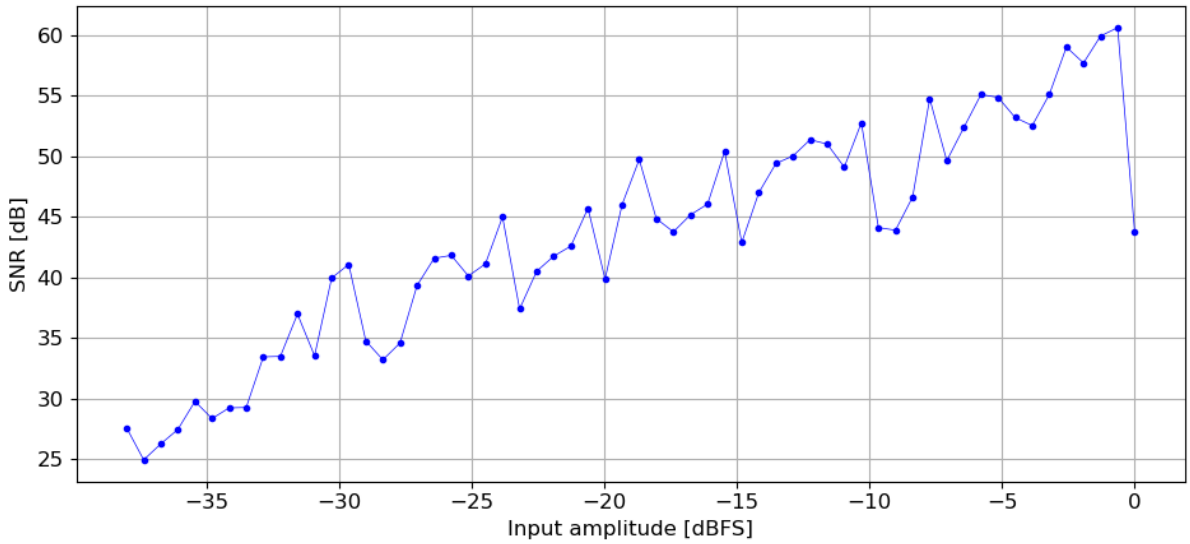


Figure 12 – SNR as a function of the input amplitude with respected to full-scale.

2.6 Higher order Sigma-Delta Modulators and MASH structures

A simple way to design a L[th] order ΣΔ modulator is to simply use equation 2.9 as its NTF, determine the loop filter transfer function H from equation 2.12 and design the loop filter to achieve such performance. For instance, the structure in figure 8 can be generalized for Lth order by using L integrators and L negative feedback in between the integrators. An important characteristic of all L[th] order modulators is that there must always be a delay in H otherwise the system will not work properly, and the output would continuously vary during the same cycle. From the previous argument, [3, pp. 95-96] shown that it that follows that H(∞) = 1.

Say we have a loop transfer function with infinite impulse response of the type

$$H(z) = \frac{b_m z^m + b_{m-1} z^{m-1} + \cdots + b_0}{a_n z^n + a_{n-1} z^{n-1} + \cdots + a_0}, \quad with\ H(\infty) = 1, \tag{2.26}$$

this means that

$$b_m = a_n \quad and \quad m = n \tag{2.27}$$

must hold when choosing a loop filter transfer function. These architectures of higher order suffer from a big stability problem which will be discussed in the next section.

Another way to implement higher order modulators is to use a multi-loop approach. A cascade of lower order modulators and some additional digital circuitry to ideally cancel out the quantization error of all the modulators except the last one, performs like a L^{th} order modulator and has the stability behavior of a second or first order one. A generalized MASH (multistage noise shaping) A+B structure is presented in figure 13.

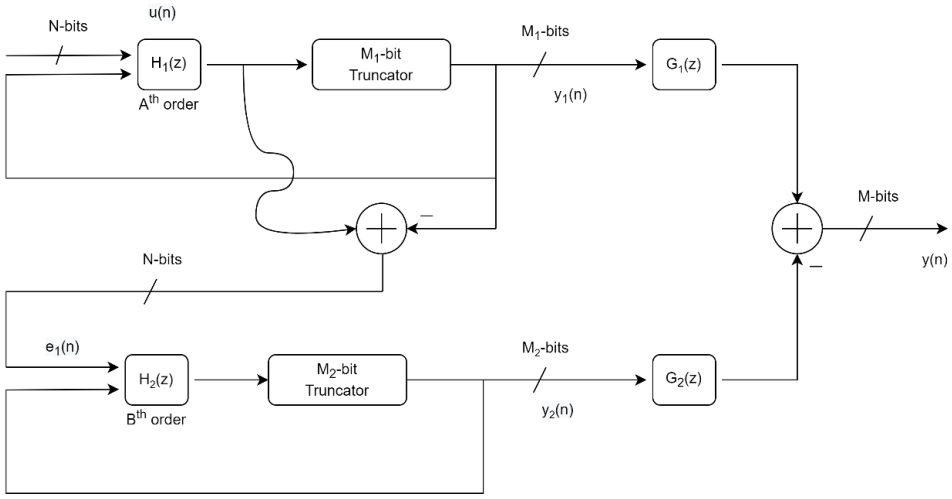


Figure 13 – MASH A+B as a $(A+B)^{th}$ order ΣΔ modulator.

The equations that describe this system are the following:

$$Y_1(z) = U(z) \cdot STF_1(z) + E_1(z) \cdot NTF_1(z) \tag{2.26}$$

$$Y_2(z) = E_1(z) \cdot STF_2(z) + E_2(z) \cdot NTF_2(z) \tag{2.27}$$

$$Y(z) = Y_1(z) \cdot G_1(z) - Y_2(z) \cdot G_2(z) \tag{2.28}$$

$$\rightarrow Y(z) = [U(z) \cdot STF_1(z) + E_1(z) \cdot NTF_1(z)] \cdot G_1(z) - [E_1(z) \cdot STF_2(z) + E_2(z) \cdot NTF_2(z)] \cdot G_2(z). \tag{2.29}$$

Remembering that the goal with this architecture is to achieve the following behavior

$$Y(z) = U(z) \cdot STF_1(z) + E_2(z) \cdot (1 - z^{-1})^{A+B} \tag{2.30}$$

and analyzing equation 2.29, it can be seen that if $G_1(z)$ and $G_2(z)$ are chosen properly, the term corresponding to the noise of the first stage can be ideally reduced to zero and the noise of the system will be only related to the noise of the second stage which will be modulated by an NTF(z) of $(A+B)^{th}$ order. Considering that $G_1(z) = STF_2(z)$ and $G_2(z) = NTF_1(z)$, the output of the system will be that of the equation 2.30.

Although this architecture has a lot of advantages when compared with single stage ones, it has the drawback of using additional circuitry for noise cancelation. Nevertheless, if this cancelation is processed in the digital domain, it can be done extremely accurately. Another drawback is that because the output is composed of the sum of two values, each with at least one bit, the output can never be single-bit.

Determining the stability limits of higher-order sigma-delta modulators is an arduous task but of great importance. Extensive high-level computer simulations may be used to understand the behavior of these systems but some rules for the effect have been proposed over the years. Lee's criterion [6] was shown to empirically yield a stable single-bit modulator requiring that the out-of-band gain of the NTF should be less than 1.5 as described in equation 2.31.

$$\max(|NTF(z)|) = \|NTF\|_\infty < 1.5 \tag{2.31}$$

This rule is not necessary nor sufficient because it says nothing about the amplitude of the input signal, and it is unlikely that it yields the best performance of the modulator. Another method used to determine stability is a root-locus approach where depending on a quantizer gain k_q, all roots must be inside the unity circle. Optimized zeros and poles values [3, pp. 107-114] of the NTF should be used in order to ensure the wanted limited out-of-band gain in output feedback structures. For N-bit OF architectures it can be shown [3, pp. 104-107] that if it has M+1 levels in the quantizer then it remains stable for inputs u(n) such that

$$\max(|u(n)|) \leq M + 2 - \sum_{n=0}^{\infty} |ntf(n)|, \tag{2.32}$$

where ntf(n) is the inverse z-transform of NTF(z). For error-feedback configurations (only suitable for digital modulators), [7] proposed that a L^{th} order (L+1)-bit modulator is stable. These architectures have the advantage that the NTF can have finite impulse response and do not need a time dependent denominator.

Even though there are a lot of rules and criterions for estimating the stability of sigma delta modulators, because these rely on a linearized model of the system, there is always a need for extensive simulation with different input amplitudes to guarantee that the chaotic behavior of these structures does not lead to instability.

2.7 Sigma-Delta Modulation in DACs

Sigma-delta modulation in analog to digital converters works practically the same as in DACs but with some minor changes. The main differences are [3, pp. 222-225]:

- In DACs, the loop configuration shown before is fully digital and does not require any type of internal data conversion.
- The digital operation in DAC sigma-delta loops removes the need for taking into consideration analog imperfections.
- Instead of the quantization performed in ADCs, DACs perform truncation of the word length, and the loop is responsible for making the signal immune to this truncation, shaping the noise out of the baseband.

Sigma-delta modulation can be used to great effect in DACs to greatly increase the SNR of the system after truncation. With N bits at the input, one can reduce this number to M, and still have the input digital signal practically untouched. One thing that has not been discussed yet is why reduce the number of bits? Let's take into consideration a 20-bit DAC. This is very hard to achieve with Nyquist DACs because the accuracy of the analog components restricts the maximum performance. By using sigma-delta modulation, one can have a 20-bit word at the input, apply it to a really simple and highly linear 1-bit DAC, and still have the performance of a 20-bit one.

Having seen how sigma-delta modulation works, in this section it will be described how it can be used in a digital to analog converter structure. The basic building blocks of a sigma-delta DAC are depicted in figure 14.

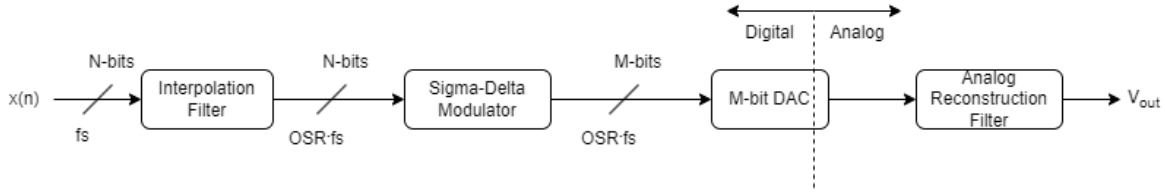


Figure 14 – Basic building blocks of a Sigma-Delta DAC.

2.7.1 Interpolation Filter

As we have seen before it is required to oversample the digital input signal. This can be achieved by using an interpolation filter which raises the input sampling rate by some value

OSR and actively removes the spectral replicas that this increase in sampling frequency introduces in the spectrum. A digital interpolation filter may be implemented by using one or a cascade of the following system of blocks depicted in figure 15.

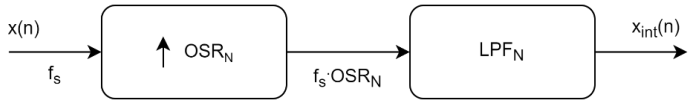


Figure 15 – Block diagram of a basic interpolation filter.

The basic principle of interpolation is that the first block performs up sampling usually by introducing OSR_N zeros in between the already existing ones, called zero stuffing. After that, the LPF removes the aliases of the signal created by the up sampling from the baseband [8]. It is good practice to design this filter to provide most of its noise suppression right after the base band of the signal where the output analog filter has most difficulty attenuating noise [3, pp. 239-243]. Most of the times, instead of performing oversampling in one single stage, a multi-stage approach is used in order to reduce the digital circuitry working at very high rates, and therefore reduce the power consumption and digital noise.

2.7.2 Single bit versus Multi-bit Truncation

After the oversampling, the signal can be driven by the sigma-delta modulator which will greatly reduce the number of bits of each sample to a value M that can be as low as 1, and therefore introduce quantization noise. This noise will be shaped out of the baseband of the signal as we have seen before. Choosing the word length after truncation is one of the tasks of the designer, with different values having different pros and cons. A truncation to a lower number of bits results in more truncation noise and a faster slew rate is required, therefore increasing the complexity of the analog reconstruction filter, but it reduces the complexity and if single bit quantization is used, it increases the linearity of the DAC. Another drawback of 1-bit quantization is that it has a poorly defined gain factor which results in stability problems. So, there is a tradeoff between linearity of the DAC, stability, and complexity of the analog reconstruction filter which must be studied carefully to achieve peak performance [3, pp. 15-16]. Furthermore, linearity problems in the analog part of the multi-bit Nyquist DAC can be attenuated by using digital circuitry to perform dynamic element matching (DEM) [5, pp. 50-53]. Table 1 breaks down the pros and cons of single bit versus multibit truncation.

Table 1 – Single bit versus multi-bit truncation.

	Single bit	Multi-bit
Linearity of the DAC	+	-
Stability	-	+
Complexity of the analog reconstruction filter	-	+
Complexity of the DAC	+	-
Increased SQNR	-	+

2.7.3 Nyquist DAC and Analog Reconstruction Filter

After truncation, the M-bit output signal of the modulator is then driven by an M-bit Nyquist DAC, and its analog output signal is then filtered by an analog reconstruction filter to remove all the shaped out-of-band noise. Conventional Nyquist DACs can be implemented in a variety of ways such as a simple changeover switch 1-bit DAC, a M-bit resistive string DAC, a M-bit R-2R DAC, a M-bit current steering DAC, a M-bit switched capacitors DAC, etc. Nyquist DACs and especially current steering DACs are covered in section 3 of this work.

The analog reconstruction filter is used with the goal to attenuate the out-of-band noise and signal aliases.

2.7.4 Sigma-Delta Modulators for DACs

As is not the case with oversampling ADCs, DAC's modulators do not require internal data conversion in the loop and every path is fully digital. This characteristic, and the fact that there is no need to account for analog imperfections in the feedback path, makes the EF configuration not only usable but highly efficient as was mentioned before. This configuration has already been shown in figure 5a. Linear analysis can show that the output is given by

$$Y(z) = U(z) + \big(1 - H_e(z)\big) \cdot E(z) \tag{2.33}$$

so,

$$STF(z) = 1 \quad and \quad NTF(z) = 1 - H_e(z). \tag{2.34}$$

For instance, setting

$$NTF(z) = (1 - z^{-1})^2 \quad \rightarrow \quad H_e(z) = z^{-1} \cdot (2 - z^{-1}). \tag{2.35}$$

This modulator is depicted in figure 16. In order to prevent modulator overflow, a limiter may be used to create saturation before wrap-around in digital values can occur.

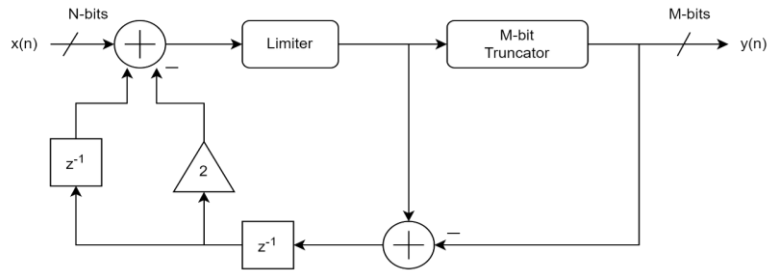


Figure 16 – Second order ΣΔ EF modulator with limiter.

This structure is only viable in DACs because the digital circuitry is sufficiently accurate to implement the loop filter. Has seen in the example of figure 16, implementing delay blocks, a gain factor of 2 and some combinatorial logic is no hard task in the digital domain. It can also be used in MASH structures.

An alternative to the error-feedback structure for M-bit truncation is to use dual truncation [9]. The main principle of this technique is to use single bit truncation on the path of the most significant bit, and M-bit truncation on the error of the first one. The most significant bit is thus converted with a highly linear DAC which improves the overall linearity of the system. Despite that, this approach requires the use of more analog circuitry, because the M-bit truncated error converted to analog must drive an analog filter to be shaped and ideally cancel out the error of the single bit truncation when added at the output. An example of this modulator is depicted in figure 17. This dual-truncation can also be used with multistage-shaping with the error of the first stage being shaped, filtered, and added at the output to ideally can cancel out the bigger error of the first stage.

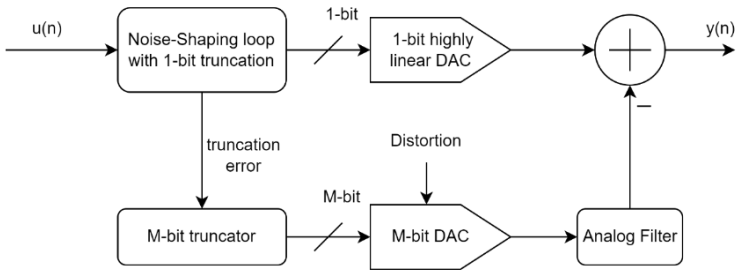


Figure 17 – Dual-truncation noise-shaping DAC.

Other techniques may be used to improve linearity and reduce element mismatch of the M-bit DAC such as data-weighted averaging, individual level averaging, vector-based mismatch shaping, tree structure element selection, digital output segmentation with scrambling and power-up calibration [3, pp. 229-238].

When it comes to output feedback there are already some general structures that are very popular and have degrees of freedom that allow various sigma-delta modulator implementations. In his book [3, pp. 115-122] and Python toolbox [10], Schreier mentions 4 architectures: the cascade of integrators with distributed feedback and input (CIFB), the cascade of resonators with distributed feedback and input (CRFB), the chain of integrators with weighted feedforward summation (CIFF), and the chain of resonators with weighted feedforward summation (CRFF). These modulator configurations are depicted in figures 18, 19, 20 and 21 respectively, for second order systems. The biggest mathematical differences between these topologies are that resonating structures, even though they are more complex, allow for the introduction of zeros not located at dc (z = 1) in the transfer function. Other than that, all topologies are very flexible.

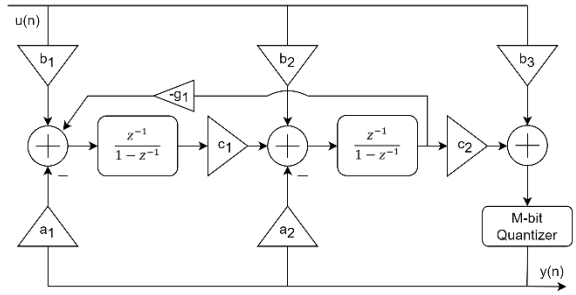


Figure 18 – CIFB configuration.

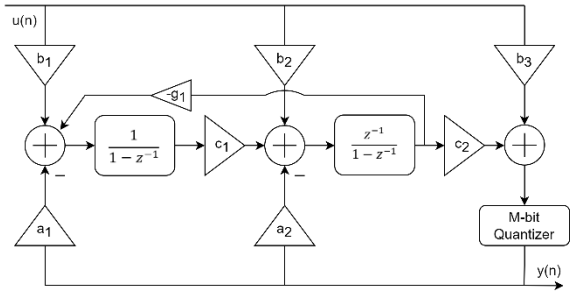


Figure 19 – CRFB configuration.

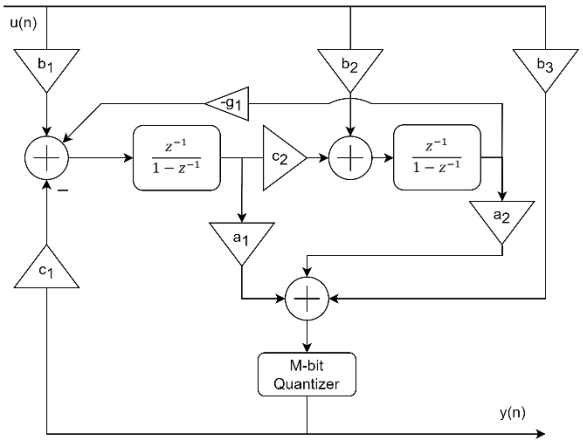


Figure 20 – CIFF configuration.

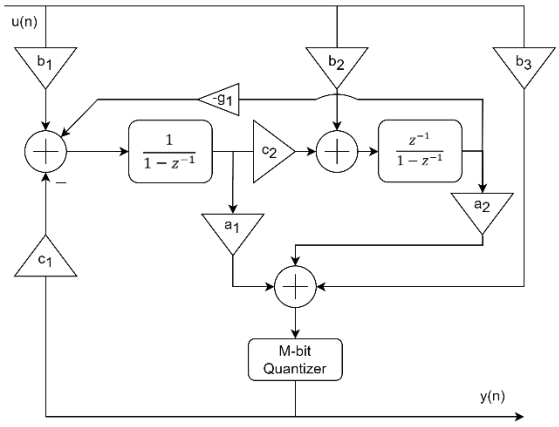


Figure 21 – CRFF configuration.

3

NYQUIST DACS

In the second chapter, sampling and quantization were discussed. In digital to analog conversion, a sampled digital quantized signal is converted to a continuous quantity like voltage or current. The input words are usually used as control bits to manipulate these quantities and reconstruct the original signal. Even though the resulting signal will be continuous, it will ideally have a single value for each clock cycle, and thus it will have a stairs shape. To fully reconstruct the original continuous signal a reconstruction filter has to be used at the output as was already mentioned in previous chapters.

3.1 DAC performance metrics

The performance metrics are the way to measure if the DAC meets the specifications of the project. Depending on the application, the most important metrics vary, but some of them are presented below. Some performance metrics for DAC where already mentioned in chapters 2.1 and 2.6 but will be repeated here for clarity.

- Differential nonlinearity (DNL) – This error is defined as the deviation of the difference of two consecutive steps and the ideal least significant bit voltage. It measures the uniformity of the steps of the transfer function of the DAC. A low DNL error is desired to achieve good resolution.
- Integral nonlinearity (INL) – This error can be defined as the cumulative some of the DNL error or the deviation between each step of the actual transfer function of the DAC and the ideal one. It is a measure of deviation of the transfer function of the DAC and a straight line. A low INL error is desired to achieve low distortion specifications.

- Gain error – This is defined as the deviation of the slope of the transfer function from the ideal slope.
- Monotonicity – A monotonic converter is one which the output always follows the increase or decrease of the input.

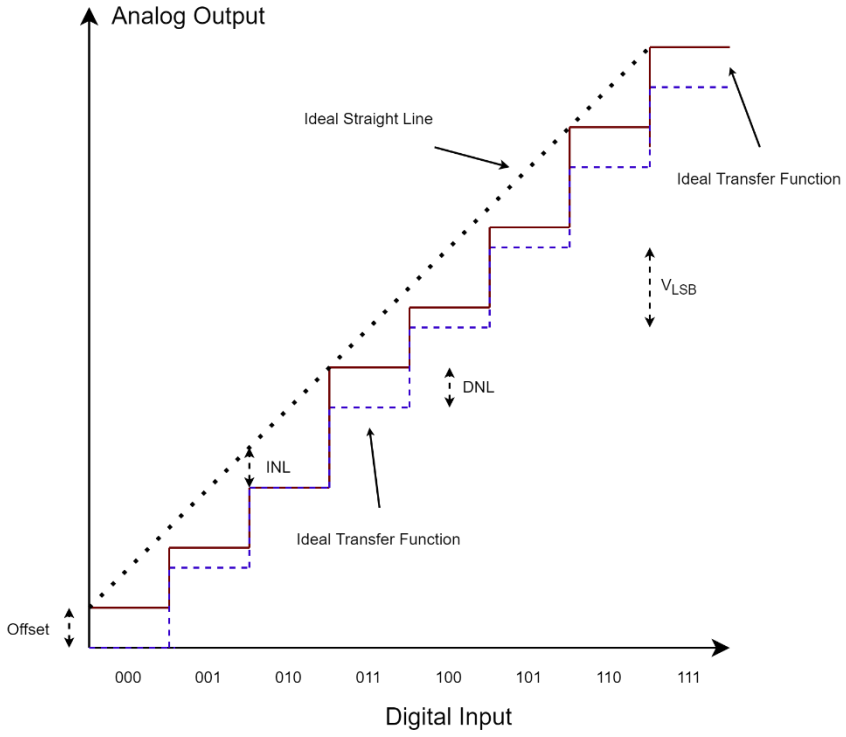


Figure 22 – DAC transfer function metrics.

- Signal-to-noise ratio (SNR):

$$SNR = 10\log_{10}\left(\frac{P_{signal}}{P_{noise}}\right) \quad [dB]. \tag{3.1}$$

- Effective number of bits (ENOB):

$$ENOB = \frac{SNR_{max} - 1.76}{6.02}. \tag{3.2}$$

- Spurious-free dynamic range (SFDR) – Is the ratio between the power of the fundamental harmonic and the strongest of the other harmonics:

$$\mathrm{SFDR} = 10\log_{10}\left(\frac{P_{signal}}{P_{strongestH}}\right). \tag{3.3}$$

- Signal to noise plus distortion ratio (SNDR):

$$SNDR = 10\log_{10}\left(\frac{P_{signal}}{P_{noise}+\sum P_{Hi}}\right) \quad [dB]. \tag{3.4}$$

- Total harmonic distortion (THD) – Is the ratio between the sum of the powers of the non-fundamental harmonics of the signal and the power of the signal:

$$SNDR = 10\log_{10}\left(\frac{\sum P_{Hi}}{P_{signal}}\right) \quad [dB]. \tag{3.5}$$

3.2 Current Steering DAC Architecture

Despite the existence of many DAC architectures, the current steering was chosen for this specific application. This topology offers high speed and with the possibility to be integrated on chip which is what this application needs. Is does not demand the use of high frequency highly linear amplifiers as other topologies would require and does not involve the charging and discharging of large capacitors at high speeds. Another advantage is that it naturally provides a differential output if needed.

3.2.1 Level of Segmentation

An important characteristic of these DACs is the level of segmentation. Current steering DACs use control digital inputs to steer current from a current source to one of two branches with resistors, creating a voltage that is proportional to the input word.

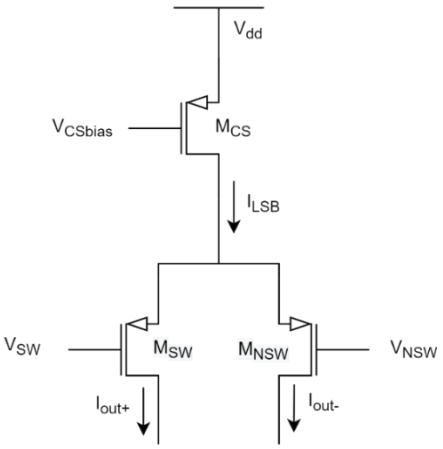


Figure 23 – Basic PMOS LSB current source with switches.

The basic current source configuration is shown in figure 23. As the input words represent binary numbers, it would be appropriate to use them to control binary weighted current sources. This would correspond to a binary implementation where for N bits, the DAC would have N current sources groups, each one with a different binary weighted current from ILSB to 2^{N-1} times ILSB. This is the easiest approach to implement but suffers from large DNL errors and doesn't always show a monotonic behaviour. This is because, for example, in a 10-bit DAC, when transitioning from code 511 (b0111111111) to code 512 (b1000000000), the group of 2^9 current sources controlled by the tenth bit would suffer a change in its state, with its current being steered from one side to another. On the other hand, all other current sources would change state in the opposite direction, which means that all bits changed value in this transition. The change of state of so many current sources at the same time is the source of these errors and leads to a static and dynamic errors. The static errors happen because each code controls a very specific group of current sources, and if it happens that, for example, one group has all current sources with a smaller current value than the desired and the other group has all current sources with bigger current value than the desired, it may not even be true that code N+1 generates a bigger current than code N, and this leads to a non-monotonic behaviour. The dynamic errors on the other hand are called glitch energy errors and happen when there are big spikes of current during transitions. Also, because each bit is always controlling the same current sources, noise, and errors in these will be correlated with the input signal, leading to input dependent errors which degrades spectral performance.

Another approach is to use a unary implementation where the input binary words are converted to a thermometer code and each value controls a single unary current source. This implementation, as the advantage that for a M code transition only M unary current sources must switch state. Going back to the previous example, when transitioning from code 511 to code 512, only one current source changes state, which is a big improvement in dynamic performance when compared to the binary implementation. Also, because there are only unary current sources, one control value does not have to be controlling the same current source for every input code. This makes this approach suited for DEM techniques that reduce input dependent errors. The disadvantage of the unary implementation is the need for a thermometer decoder which increases complexity, power consumption and area of the circuit.

To take advantage of both implementations, the most used approach is a segmented one which can be seen in figure 24. Generally, the K least significant bits controlling binary weighted current sources and the N-K most significant bits controlling unary current sources

after being converted to thermometer code. The choice of how many bits to use for each approach is a very important one for the design of the DAC. Segmentation is generally referred as the percentage between the number of thermometer coded bits and the total number of bits. Literature up until now suggests that segmentation from 60% to 70% gives the best tradeoff between power, area, and dynamic performance of the DAC [12].

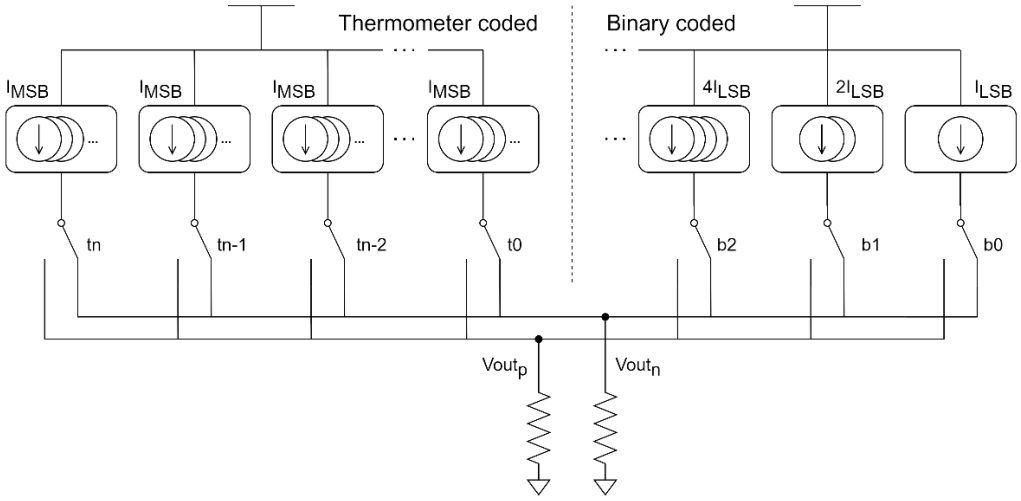


Figure 24 – Segmented implementation of differential CSDAC.

3.2.2 Current Source

In this section, the PMOS version of the current source and switches have been used to demonstrate important characteristics of this block. Even though they are slower than the NMOS counterpart, the use of PMOS transistors is preferable due to the fact that the devices shield the output node from VDD, improving power supply rejection ratio, and thus reducing noise from the power supply at the output node.

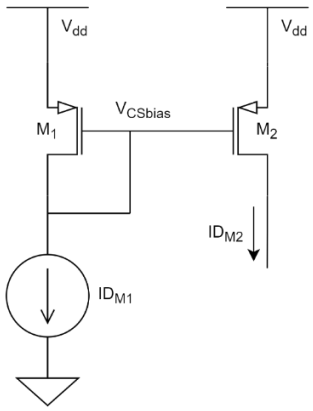


Figure 25 – Basic current mirror.

As was seen in figure 23 the current source block is composed of a transistor designed to mirror the current of a biasing circuit and two switches in a cascode configuration. These should be sized to be in saturation region in order to obtain the best mirroring and improved intrinsic resistance. The basic current mirror structure is presented in figure 25. Considering all transistors to be working in strong inversion and saturation region, the drain current can be approximated by the quadratic model and is given by

$$I_D = \frac{1}{2}\mu C_{ox}\frac{W}{L}(V_{GS} - V_{TH})^2(1 + \lambda V_{DS}), \tag{3.6}$$

where μ is the mobility constant, C_{OX} is the oxide capacitance, W is the width and L is the length of the transistor, V_{GS} is the gate-source voltage, V_{TH} is the threshold voltage, λ is the output impedance constant and V_{DS} is the drain-source voltage. Figure 25 shows that if the transistors have the same technology parameters, and because the gate-source voltages are forced to be equal, the ratio between the drain currents of these two devices will be

$$\frac{I_{D_{M1}}}{I_{D_{M2}}} = \frac{\left(\frac{W}{L}\right)_{M1}\left(1 + \lambda V_{DS_{M1}}\right)}{\left(\frac{W}{L}\right)_{M2}\left(1 + \lambda V_{DS_{M2}}\right)}. \tag{3.7}$$

Generally, λ is very small with its value being inversely proportional to the length of the device, and the ratio can be approximated to be the ratio between the W/L of each device. This is a good approximation but for small channel devices, one must be careful when not considering this parameter. The drain-source voltages of the devices should be as close as possible to each other, and these devices should have L as high as possible to minimize the channel length modulation effect.

Another important characteristic of the current source is its output impedance. Having a finite value, it will be seen in parallel with the conversion resistor at the output. Furthermore, because of the nature of the current steering converter, the output not only sees one current source output impedance but N all in parallel, with N being the input code. This leads to a code dependent output impedance which creates an input dependent output voltage and degrades the spectral performance of the converter as was pointed out by [13]. This author suggests reducing this erratic behaviour by using a differential output. This comes with little to no cost for a current steering DAC as both outputs are already available. Author [14] suggests the use of an extra cascode transistor either at the drain of the current source or at the drain of

the switches to increase the output impedance. These can be seen in figure 26 a) and b) respectively. The first approach requires only one extra device per current source, and it has the extra advantage of isolating the common node CN from the current source device which should have the biggest parasitic capacitances, reducing the glitch energy caused by the common node voltage variation. Also, with the use of the cascode transistor, the V_{DS} of the current source can be set relatively accurately by V_{GS} of the cascode, improving mirroring and guaranteeing that the device remains in saturation. The second approach provides isolation of the output node from the switch transistors, reducing digital input feedthrough through the gate-drain capacitance of the device. Also, both improved versions of the current source improve output impedance frequency response.

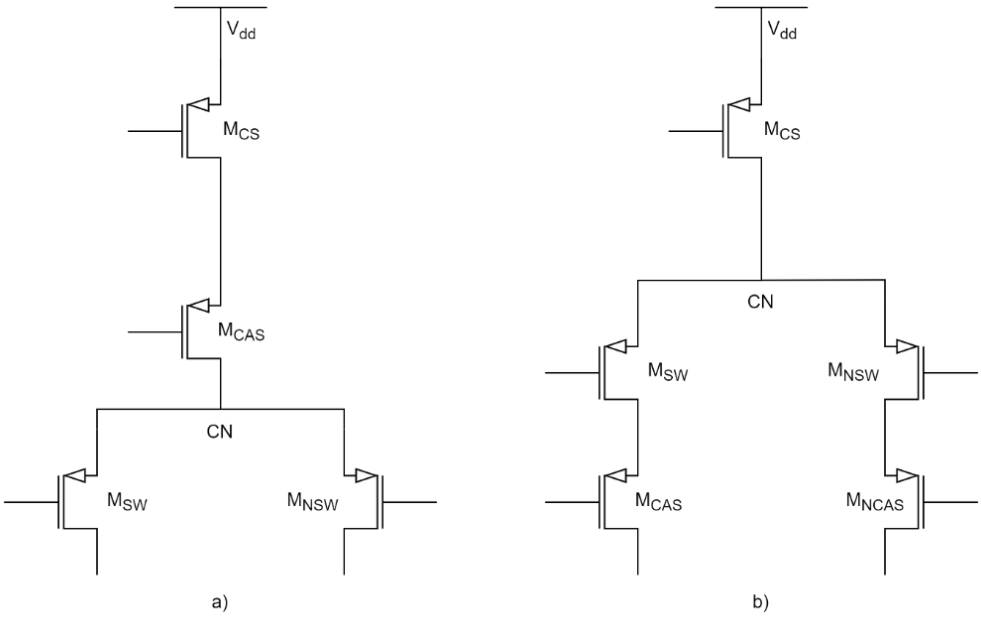


Figure 26 – a) Cascoding the current source and b) cascoding the switches.

3.2.3 Control Logic Blocks

As has been discussed before, when using a segmented implementation of a CSDAC, there is a need for a binary to thermometer decoder for the unary part of the current sources. This is one of the digital blocks the signal goes through before controlling the steering of the current sources. Because this decoder as some intrinsic delay between the input and the output, and if nothing is done to the binary bits, there would be great asynchronism between the two control systems. To mitigate that problem, a latency equalizer must be used in the binary part of the control system. This block could be implemented by an even chain of inverters that

would delay the binary control codes ideally by the same time it takes the thermometer decoder to process the other bits.

Even though these blocks would be a great step towards good synchronous control, they are not enough. There is a need for a latch and driver before the control signal can effectively steer the current sources. This block overcomes glitch and distortion and improves dynamic performance as stated in [15]. The main goal is to guarantee that both switches are never completely off at the same time to avoid big settling times due to charging and discharging of the parasitic capacitances at the common nodes. In the case of PMOS switches the device is active-low, and in the most basic case it is turned on by applying zero volt to its gate generating a source-gate voltage bigger than the threshold voltage. The complementary control signals applied will have to cross each other to steer the current from one side to another and to achieve the goal of never turning off both switches at the same time this crossing point must be lowered. An example of this is showed in figures 27 and 28.

Figure 27 - Crossing point midway through the switch control voltage range.

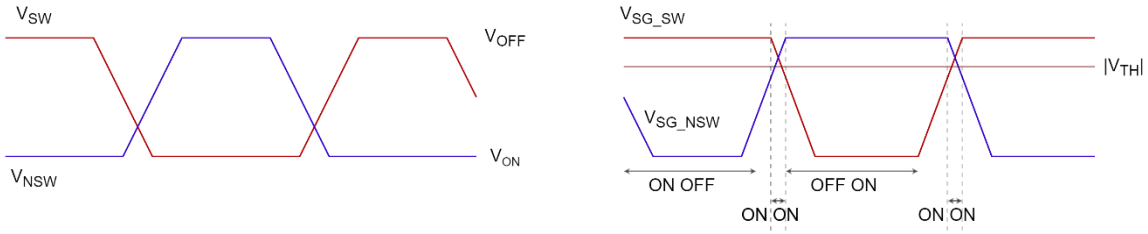


Figure 28 – Crossing point lower than midway through the switch control voltage range.

The approach shown in figure 28 is just one that can be used to lower the crossing point and it is used here for its visual simplicity, but there are other ways to perform this task. Reference [16] suggests the use of different rise and fall times for the control wave but one could make use of the fact that the control signals of the switches do not have to be VDD for off state and GND for on state. Actually, they just need to be low enough to get a $V_{SG} > |V_{TH}|$ and high enough to get $V_{SG} < |V_{TH}|$, respectively. This reduction of the range of the control signals is

beneficial because it improves dynamic range at the output, reduces control signal feed-through and makes the manipulation of the control signals to get a lower crossing point much easier. For instance, if a crossing point lower than 400 mV is needed, a high voltage of 700 mV and a low voltage of 100 mV with perfect phase opposition would fulfil this need. Of course, this is only valid if the high voltage is enough to turn of the switch and the low voltage enough to turn it on. A level shifter to address these issues that must be used. This would be a simple circuit that limits the input wave form to certain voltages. Despite that, these voltages are not simple to generate, and that is disadvantage of using this approach.

3.2.4 Static Behaviour

The static performance of the DAC is mostly determined by matching accuracy between individual LSB current source transistors. This mismatch leads to a random variation in the INL which is important to predict within certain boundaries. Author [17] refers 3 methods that give a relationship between the current source relative standard deviation and the INL yield. The INL yield is defined as the probability that a CSDAC has an INL lower than a certain value. Generally, the maximum INL value for a well-built CSDAC is half the LSB to achieve a monotonic behaviour and ensure that the maximum non-linearity error is smaller than the maximum quantization error [18]. The 3 methods mentioned above are:

- The Lakshimikumar approach [19], which was the first attempt to analytically determine the INL yield of a converter and assumes that all output codes are uncorrelated.
- The Monte Carlo approach, which uses extensive numerical simulation to get a statistical measurement of the converter.
- The New INL_Yield Formula, which is given by:

$$\frac{\sigma(I)}{I} \leq \frac{INL_{max}}{C \cdot \sqrt{2^N}}, \tag{3.8}$$

$$C = inv_{norm_{(-x,+x)}}\left(0.5 + \frac{INL_{Yield}}{2}\right),$$

where N is the resolution of the converter and the function used to calculate C is the inverse normal cumulative function integrated from -x to x. The python scipy.stats library offers this function as norm.ppf().

After determining the value of the maximum relative standard deviation, one must use that value to properly size the current source transistor. The Pelgrom model [20] is the most widely used and states that the area of this device is given by equation 3.9,

$$W \cdot L \geq \left(\frac{4{A_{V_{TH}}}^2}{(V_{GS} - V_{TH})^2} + {A_\beta}^2 \right) \Big/ \left(\frac{\sigma(I)}{I} \right)^2, \tag{3.9}$$

where A_{VTH} and A_β are technology dependent matching constants related with variations in the threshold voltage and the constant $\beta = C_{ox}\mu W/L$. So, for a higher required matching, a bigger device must be used, and bigger gate-source voltages is helpful in reducing the area needed to achieve a certain matching.

3.2.5 Dynamic Behaviour and High Frequency Performance

As pointed out by reference [14] the dynamic performance of the CSDAC is mostly influenced by timing and switching errors, settling time, capacitive feedthrough from the control signals to the output, transistor state output voltage dependency and finite and code dependent output impedance. Some of these issues are depicted in figure 29.

Timing and switching errors are prejudicial because the switches do not reach the goal operation region at the same time. These errors create a change in the voltage of the common nodes which charges and discharges the parasitic capacitances, limits bandwidth and dynamic performance. As has been mentioned, in the worst-case scenario, both switches will be off at the same time. This should be avoided because the common node voltage would discharge drastically and, afterwards, when one switch turns on again, the node capacitance would need to be charged again, stealing current from the output to this end. These changes in the transient voltage of the common node also affect the biasing of the cascode and current source transistors through the gate-drain capacitance and, ultimately, affect the output current value.

The capacitive feedthrough of the control signal to the output leads to an output voltage variation given by

$$\Delta V_{out} \approx V_{SW_{Swing}} \frac{C_{GD_{SW}}}{C_{out_{total}}}, \tag{3.10}$$

a capacitive voltage divisor between the gate-drain parasitic capacitance and the total output capacitance. This means than to reduce capacitive feedthrough one could reduce the switch size to reduce the parasitic capacitances or lower the switch control voltage swing.

A variation of the transient voltages of the common node can also occur with the change of the output voltage. With this, there is also a change in the state of the transistors, from moderate to deep saturation and, therefore, a change in current and output impedance. So, even if there is a guarantee that all transistors remain in saturation, with the swing of the output voltage, the output impedance does not remain constant and might change 10% to 20%. Moreover,

the output impedance is doubly affected by the input digital signal because its value depends on the number of current sources that are switched on and connected to that output node, as the individual unary current cell output impedances will be in parallel with each other and the conversion impedance. Assuming that all current sources have the same output impedance, the actual conversion impedance is given by

$$Z_{real}(n) = Z_L \parallel \left(\frac{Z_{out}}{n}\right), \tag{3.11}$$

where Z_L is the conversion impedance at the output and, Z_{out} is the current source output impedance and n is the input code which is the number of current sources connected to the output node.

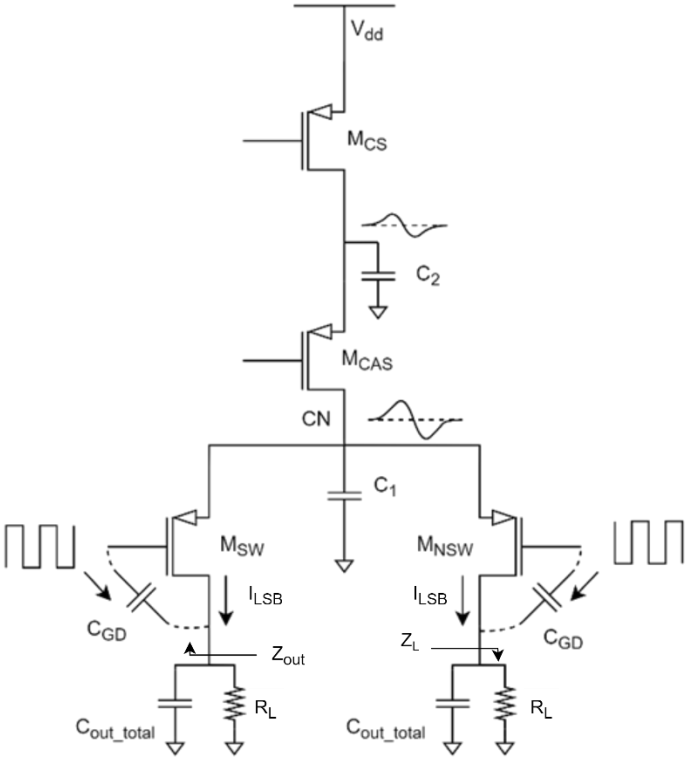


Figure 29 – Unary current cell dynamic behaviour.

The effects of finite code dependent output impedance were first studied in [14] where the second (HD2) and third order distortion (HD3) power for single ended and fully differential outputs respectively were derived to be:

$$HD2_{SE} \approx 20\log_{10}\left(\frac{NR_L}{4|Z_{out}|}\right)\ [dBc], \tag{3.12}$$

$$HD3_{FD} \approx 20\log_{10}\left(\frac{N^2{R_L}^2}{16|Z_{out}|^2}\right)\ [dBc]. \tag{3.13}$$

Reference [14] also presents a relationship between the output impedance and the achievable INL. This is given by equation 3.14.

$$INL = \frac{I_{LSB}{R_L}^2 N^2}{4|Z_{out}|} \tag{3.14}$$

Due to the parasitic capacitances C_1 and C_2 shown in figure 29, the output impedance of the current cell degrades as the signal frequency is increased. Reference [14] shows that the small signal analysis yields an output impedance over frequency equal to equation 3.15 for non-cascode and 3.16 for a cascode implementation of the current cell.

$$Z_{out\_noCAS} \approx \frac{gm_{SW}}{gds_{SW} \cdot gds_{CS}} \cdot \left(\frac{1 + j\omega C_1/gm_{SW}}{1 + j\omega C_1/gds_{cs}}\right) \tag{3.15}$$

$$Z_{out\_CAS} \approx \frac{gm_{SW} \cdot gm_{cas}}{gds_{SW} \cdot gds_{CaS} \cdot gds_{CS}} \cdot \left(\frac{1 + j\omega C_1/gm_{cas}}{1 + j\omega C_1/gds_{cs}}\right) \cdot \left(\frac{1 + j\omega C_2/gm_{SW}}{1 + j\omega C_2/gds_{cas}}\right) \tag{3.16}$$

So, it is expected that the output impedance starts to drop off at frequency equal to its dominant pole.

3.2.6 Thermal and Flicker Noise

In a Nyquist DAC, thermal noise typically would be below the quantization noise level and would not even be considered but in an oversampled DAC, the oversampling may lower the quantization noise floor to a point in which the thermal noise may became dominant. In a cascode configuration, it is known that the noise of the cascode device is much lower than the common source transistor and it may be neglected. Considering that all noise sources are uncorrelated, adding the current noise powers of all N current sources and converting to voltage noise power by multiplying the by the output node impedance R_L squared, the in-band output thermal noise power spectral density is approximately given by

$$S_{th_{DAC_{out}}} \approx N \cdot 4K_B T\gamma gm \cdot {R_L}^2 \quad [V^2/Hz], \tag{3.17}$$

where K_B is the Boltzmann constant, T is the temperature in Kelvin, γ is a technology parameter that depends on the channel length, and gm is the current source device transconductance.

Another noise source that is important to consider is flicker noise. Flicker noise is also known as 1/f noise because its power spectral density is inversely proportional to frequency. In a MOSFET, the input referred flicker noise PSD can be derived as

$$S_{f_{in}}(f) = \frac{k_f}{C_{ox} \cdot W_{eff} \cdot L_{eff} \cdot f^{A_f}} \quad [V^2/Hz], \tag{3.18}$$

where k_f and A_f are technology dependent parameters, C_{ox} is the oxide capacitance per unit area, and W_{eff} and L_{eff} are the effective channel width and length of transistors, respectively [26]. For this application it will be considered that A_f is approximately 1 and the other constants will be put together in only one constant named K. Translating this noise to the output of the DAC and summing the contribution of all current sources, the in-band output flicker noise power spectral density is approximately given by

$$S_{f_{DAC_{out}}}(f) \approx N \cdot \frac{K}{f} \cdot gm \cdot {R_L}^2 \quad [V^2/Hz], \tag{3.19}$$

with

$$K = \frac{k_f}{C_{ox} \cdot W_{eff} \cdot L_{eff}}. \tag{3.20}$$

4

HIGH-LEVEL MODELLING OF A SIGMA-DELTA DAC THAT DRIVES A VCO

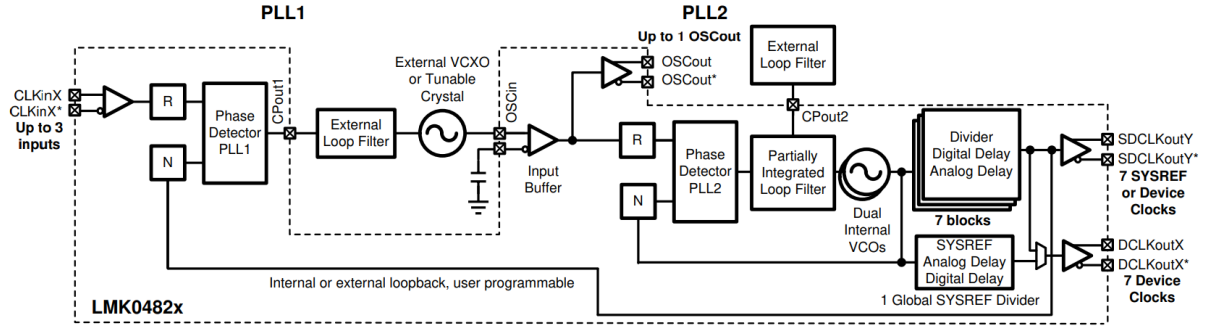
Generally, designers use a hierarchical synthesis methodology to implement Sigma-Delta converters. One of the first stages of this methodology is high-level modeling where the behavior of this chaotic system with non-linear data is simulated numerically and exhaustively to find the right synthesis that fulfills the specifications and yields a robust stable implementation. Only then electrical and circuit level simulations should be done as a last validation, otherwise it would take too long to fine tune the modulator. The other blocks of the sigma-delta DAC can also be high-level modelled to understand if their non-idealities impact the overall performance of the system, and most of the time they do, especially the analog part.

4.1 Context in the PLL Application

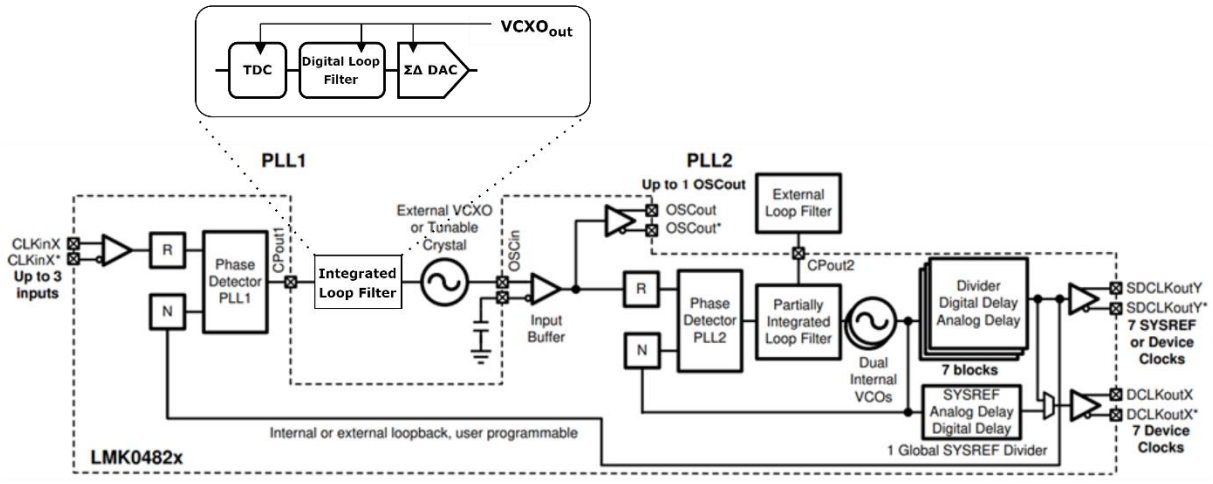
As was mentioned in the introduction of this work, this project is inserted in a PLL application [11] and the main goal of the DAC is to be part of an integrated loop filter that will substitute the external loop filter that perform extremely accurate voltage control of a VCXO in PLL1 and whose block diagram is in figure 30 a). The substitution to be made can be seen in figure 30 b). The external analog loop filter is to be substituted by an integrated digital loop filter with a time to digital converter (TDC) driving it and the ΣΔ DAC following it to converter back the digital words to an analog control signal of the VCXO.

A PLL is the standard system used to generate a signal that follows some input frequency. In the case that the desired signal has to be very pure, a crystal oscillator (XO) may be used as reference because of its inherent low noise. But the application may not want to follow

the XO nominal frequency, but rather some other reference frequency. So, the goal is to use a system that has the absolute frequency of some noisy input and the noise performance of the XO. For that, the double-loop PLL architecture is used. In figure 30 two loops can be seen, the inner loop PLL2 and the outer loop PLL1. PLL2 is a conventional PLL architecture, and its input is the being generated by a very low noise VCXO. The VCXO is part of the outer loop which runs at very low bandwidth and has as input the frequency the reference frequency. The slow response of the outer loop compared with the inner makes it so that the inner loop works practically independently of the outer one and the VCXO frequency is slowly adjusted to get the correct low noise output signal.



a)



b)

Figure 30 – a) Original b) Modified to have an integrated loop filter - Texas Instruments LMK0482x Clock Jitter Cleaner diagram for nested zero-delay dual-loop mode [11]

So, the first PLL is responsible for cleaning the input jitter. Also, instead of the first PLL having its own output fed back to the input, it's the full system's output which is fed back. This causes the clock outputs to have deterministic phase relationship with the clock input [11].

4.1.1 Phase noise in Voltage Controlled Oscillators

Noise is a major concern in oscillators, and it may lead big changes in its frequency spectrum and timing accuracy. In the time domain, timing inaccuracies, meaning a deviation of the oscillating period from its ideal value, are called timing jitter. There are various types of jitter but generally, the most used definition is called cycle jitter or rms cycle jitter and measures the variance of each period from the average one [23] [24]. Phase noise is the frequency representation of jitter and rms cycle jitter and the phase noise power spectral density can be related by equation 4.1, where S_θ is the phase noise PSD, f_1 and f_2 are the bounds of the measured phase noise spectrum and f_0 is the oscillator's free-run frequency.

$$\sigma_j^2 = \frac{\sqrt{2 \cdot \int_{f1}^{f2} S_\theta \, df}}{2\pi f_0} \tag{4.1}$$

One of the specifications for the DAC is to produce a lower phase noise at the output of the VCXO than the VCXO itself. Reference [21] shows that, in a voltage-controlled oscillator (VCO), the voltage noise from the control signal at the input is converted to a phase noise at the output. The basic system equation for a VCO is in 4.2 where f0 is the free-run frequency of the oscillator, Kv is the gain of the VCO with units of Hz/V and vin is the control input voltage. The second term of the sum inside the cosine is called excess phase and is represented by equation 4.3.

$$VCO_{out}(t) = A_0 \cos\left(\omega_0 \cdot t + K_v \int v_{in}(t)\, dt\right) \tag{4.2}$$

$$\theta_{ex}(t) = K_v \int v_{in}(t)\, dt \tag{4.3}$$

So, the output phase noise $\Delta\theta$ can be related to the input voltage noise Δv by equation 4.4. where Sv is the input noise power spectral density and Δf is the frequency offset from the carrier.

$$\Delta\theta(t) = K_v \int \Delta v(t)\, dt \quad \overset{PSD}{\Longrightarrow} \quad S_\theta = \frac{{K_v}^2 \cdot {S_v}^2}{\Delta f^2}\ [1/Hz] \tag{4.4}$$

A representation of the modulation shown by equation 4.4 is depicted on figure 31. The VCO acts like a perfect integrator with a pole at $\Delta f = 0$ and attenuates the noise by 20 dB per decade.

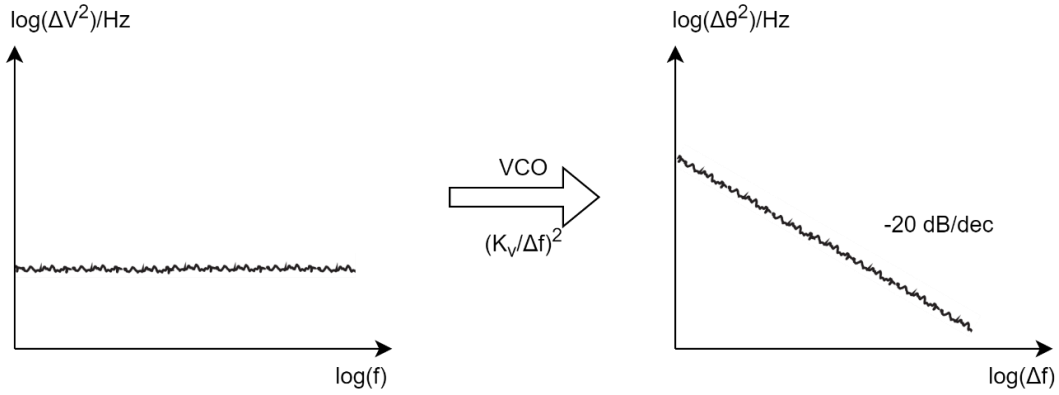


Figure 31 – VCO input white noise to output phase noise

Reference [22] shows that variations in the input controlling signal of the VCO (say sinusoidal variations) may be frequency modulated by the VCO and lead to unwanted spectral components at the output. Choosing $v_{in}(t) = A_m \cos(\omega_m t)$, using a few trigonometric relationships and considering A_m small enough that $K_v A_m/\omega_m \ll 1$ we get that

$$VCO_{out}(t) = A_0 \cos(\omega_0 t) - \frac{A_0 K_v A_m}{2\omega_m}[\cos((\omega_0 - \omega_m)t) - \cos((\omega_0 + \omega_m)t)]. \quad (4.5)$$

Therefore, when the input control signal possesses variation with time, the output will have two unwanted components with normalized amplitude equal to equation 4.6 at the carrier frequency plus or minus an offset which is the frequency of the noisy input signal.

$$\Delta\theta_m = \frac{K_v A_m}{2\omega_m} \quad (4.6)$$

This also means that, when the VCO operates in steady state, its input must experience very slight variations as to not corrupt the output signal. The expected output spectrum for the example presented above is in figure 32.

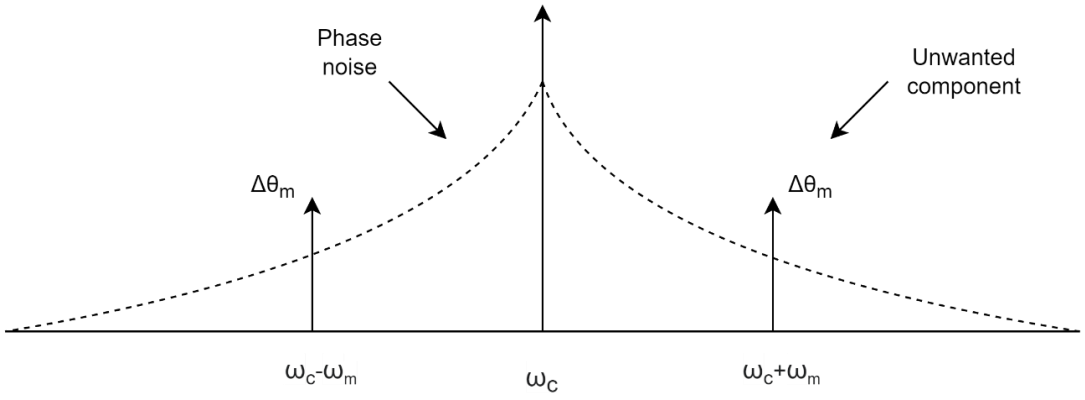


Figure 32 – Phase noise at VCO output.

4.1.2 Project Specifications

Figure 30 shows the block diagram of the double PLL. One of the blocks, the loop filter of the first PLL was initially designed as an analog filter external to the chip and is to be substituted by a digital loop filter that outputs digital words of 48 bits. Then, this signal needs to be converted to an analog voltage to control the VCXO, and that's the purpose of the sigma-delta DAC. The specifications for this sigma-delta DAC are presented in table 2. Looking at it and using equation 2.7, the oversampling ratio of this system is 61440 which is a huge value. This is achievable because, as explained before, the bandwidth of the first PLL is very narrow (1 kHz), and the clock sampling frequency of the system is equal to the free-run frequency of the VCXO because it is being taken advantage of to create the clock for the digital filter.

The purpose of this section 4 is to create a high-level model of a sigma-delta DAC and take conclusions about the structure and architecture which fulfills the system requirements in table 2.

Table 2 – Sigma-delta DAC and system specifications.

Specification	Notes	Typical Value	Units
F_0	VCXO clock free-run frequency	122.88	MHz
K_0	VCXO tuning sensitivity	25	ppm/V
K_v	VCXO gain ($K_0 \cdot F_0$)	3072	Hz/V
w_{in}	DAC input word length	48b	Signed
Phase noise @ f_m = 30 Hz	Referred to VCXO output	-100	dBc/Hz
Phase noise @ f_m = 100 Hz	Referred to VCXO output	-114	dBc/Hz
Phase noise @ f_m = 1 kHz	Referred to VCXO output	-138	dBc/Hz
Phase noise @ f_m = 10 kHz	Referred to VCXO output	-153	dBc/Hz
Phase noise @ f_m = 100 kHz	Referred to VCXO output	-161	dBc/Hz
Phase noise @ f_m = 1 MHz	Referred to VCXO output	-166	dBc/Hz
Phase noise @ f_m = 10 MHz	Referred to VCXO output	-166	dBc/Hz
SFDR	Referred to VCXO output	100	dBc
BW	DAC bandwidth	0.1 to 1k	Hz
F_s	Clock sampling frequency	122.88	MHz

4.2 Model Overview

The overall block diagram of the sigma-delta DAC was shown in figure 14. The first block, counting from left to right, is the interpolation filter that up samples the signal to the desired OSR. In this application there is no need for such block because the input signal to the modulator is already sampled at 122.88 MHz. The model presented here will only contemplate the modulator, the DAC, and the reconstruction filter. Figure 33 gives an overview of the 3 blocks, their inputs and outputs, and the input and output data processing that is needed.

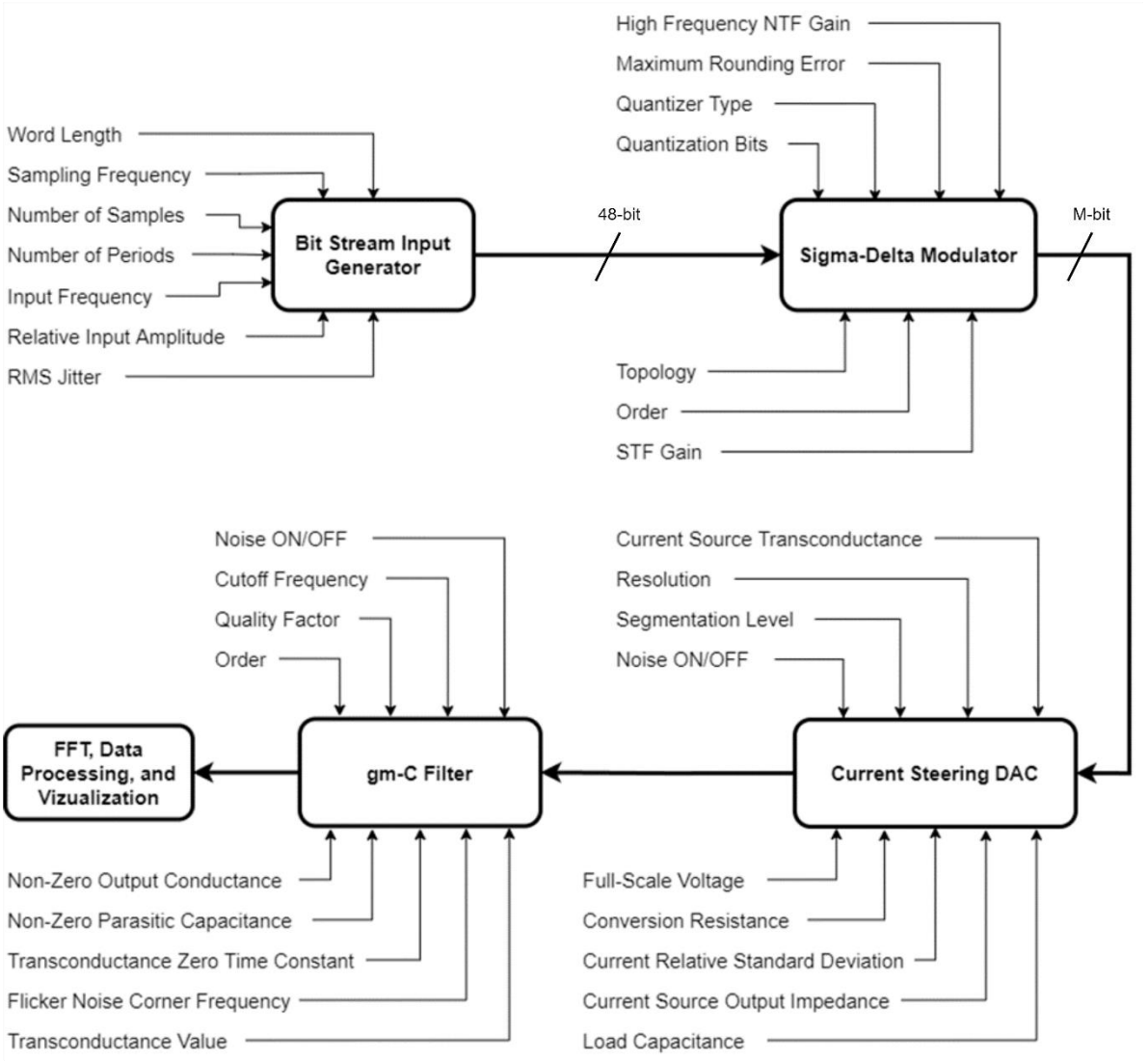


Figure 33 – Block diagram of the modelled system.

The modulator can be configured to have different orders, to have a 1-bit to M-bit quantizer, and to have the basic output feedback modulator (OFM), error feedback modulator

(EFM) and MASH topologies, and other more complex topologies like the ones shown in section 2.10.4, whose coefficients can be optimally calculated using the Schreier Toolbox.

When simulating with a modulator that outputs more than 1-bit, the DAC chosen for this application, and therefore modelled, was a current steering DAC. The model considers all the main non-idealities mentioned in section 3 and the noise associated. This choice was based on the following considerations:

- The speed requirement in relatively high and CSDACs are known to be a good solution for high-speed conversion.
- The purpose of the substitution of the external filter by a digital system is to remove external components and integrate as much as possible on chip. For that, CSDACs are also a good choice because they do not involve the use of big resistors that are hard to integrate with good accuracy.

The last block modelled was the analog reconstruction of filter. Its goal is to attenuate the shaped high frequency noise created by the modulator and to act as an anti-aliasing filter, smoothing the output of the DAC. The cutoff frequency is at least equal to the bandwidth of the system, but it can be higher as long as the shaped high frequency noise does not corrupt the signal more than the acceptable values. Its requirements are relaxed by the high oversampling ratio used in the system as the spectral images are very further apart in the spectrum and should not be difficult to remove them. Also, this filter should be an analog continuous filter to avoid further aliasing. Even though there is a clock already available in the system and it samples at a much higher rate than the bandwidth of the first PLL (so aliasing would not be a problem), the second PLL has a significantly wider bandwidth and using a discrete filter may be problematic. It would probably require the use of an external RC filter at the output anyway, to remove spectral images, which is against the goal of not using external to chip components. One suggestion for this filter would be a gm-C implementation, a continuous filter which substitutes the use of resistors by transconductances. The easiest way to model a filter is through its transfer function and that's the approached used in this work. The drawback of this approach is that to use a transfer function of a system, it must be considered linear and time invariant and, therefore, non-linearities of the filter are not taken into consideration in this model. This is a considerable drawback since one of the major flaws of gm-C filters is its high non-linear behaviour due to the use of transconductance amplifiers. Another major drawback of this approach is that for good noise performance at such low cutoff frequency, big capacitors may be needed which may be against the on-chip goal. If the SFDR or noise

performance of the gm-C reconstruction filter are not enough to meet the requirements, a switched capacitor filter may be considered, or we may arise at the conclusion that an on-chip solution is not feasible. So, from a system point of view, the reconstruction filter seems to be the critical block.

4.3 Sigma-Delta Modulator Model

Section 2.6 shown an example of discrete time model used to simulate the first order modulator in figure 6. The mathematical operations in equation 2.24 can be implemented in most programming languages, and in this case, Python was used. Integer values were used instead of binary strings for simplicity. Addition, subtraction, and multiplication are elementary operations to perform.

The truncation operation was achieved by right shifting. Say the goal is to truncate the number N which has 48 bits to its 10 most significant bits. Using the right shift operator >> this can be achieved by

$$N_{trunc} = N \gg (48 + 1 - 10). \tag{4.7}$$

One detail that is important not to overlook is that the number being truncate might not have the same number of bits as the input. In the example of equation 2.24, the number being truncate is the summation of 3 other numbers and depending on these values the biggest number to be truncated would actually have one more bit than the input. That is why in equation 4.7 there is an addition of a 1 in the right shift term. Another important detail is that the value that is fed back cannot be the 10-bit number (in this example) that would be subtracted from the 48-bit input. To feed back the output, it must be resized to the size of the input and for that a left shift is used. So, the value that is fed back is actually the value before truncation but with all the least significant bits set to zero.

Other important operation in these systems is the time shift because the computation a value always depends on previous computed values. For that, arrays are used and as the values are computed (starting from some initial conditions) they are saved in a position of the array. If the next value is computed using a previous one, it can be accessed easily.

These are models that are supposed to represent actual digital electronic devices that may have a cap on the number of bits they can work with. As mentioned in section 2.10.4, a limiter may be used to prevent big calculation errors due to overflow. Especially in this case

that all signals being processed are signed (meaning they can be positive and negative), and because this is handled by using a two's complement binary number, if overflow occurs, one very large positive number may be interpreted as a small negative number. This example is for a basic first order OFM, but the methods can be extrapolated for other architectures.

The models created in this work are for EFM1, EFM2, OFM1, OFM2, MASH2 and MASH3, CIFB2, CRFB2, CIFF2, and CRFF2, where the coefficients of the last for are computed with the help of the Schreier sigma-delta library. The block diagrams of these modulators are shown in figures 34 to 43. The number following the acronym of the topology refers to its order. One additional topology that was modelled is the first order hybrid-key error feedback modulator (HKEFM1) which is a modified version of the EFM1 and is used in the MASH topologies. The value of the coefficient "a" is chosen so that it maximizes the sequence length of the modulator, working as a dithering process which smoothens its response.

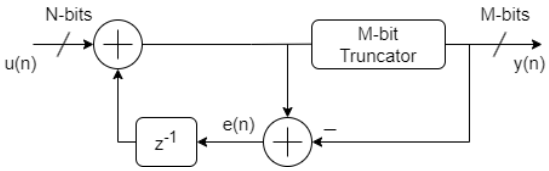


Figure 34 – EFM1.

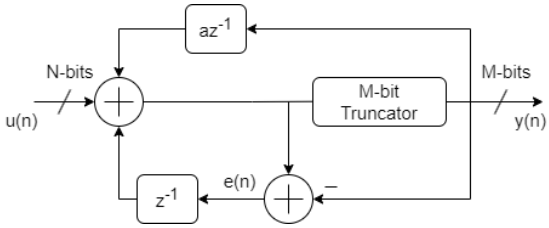


Figure 35 – HKEFM1.

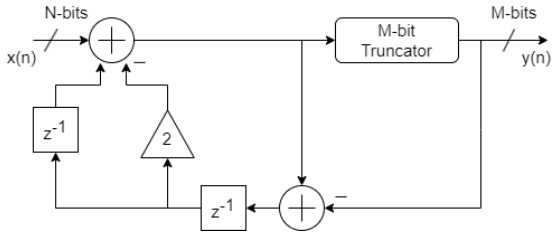


Figure 36 – EFM2.

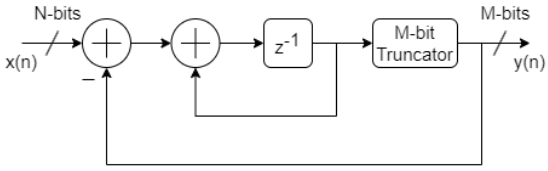


Figure 37 – OFM1.

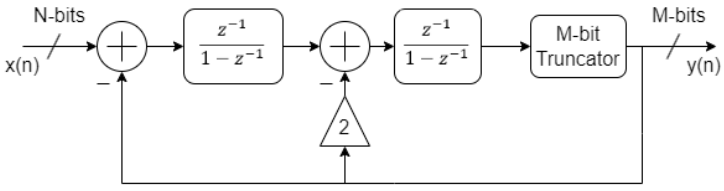


Figure 38 – OFM2.

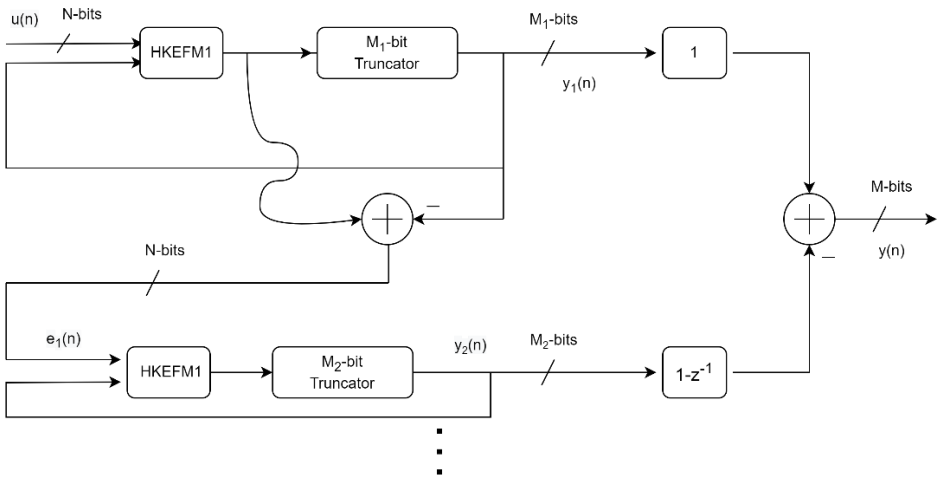


Figure 39 – HKEFM1 MASH.

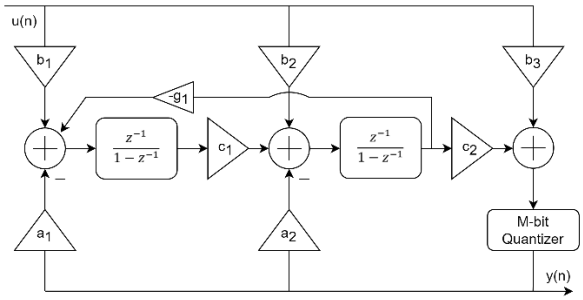


Figure 40 – CIFB2.

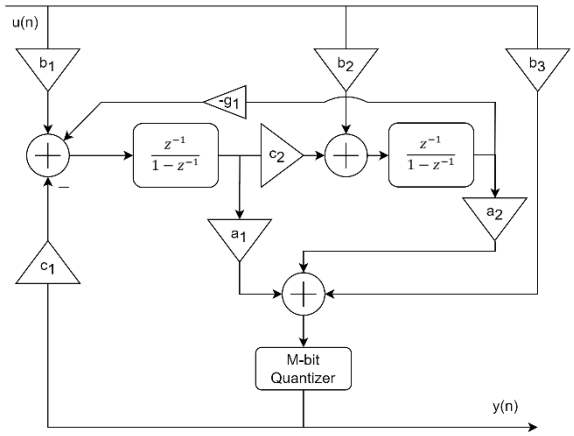


Figure 41 – CIFF2.

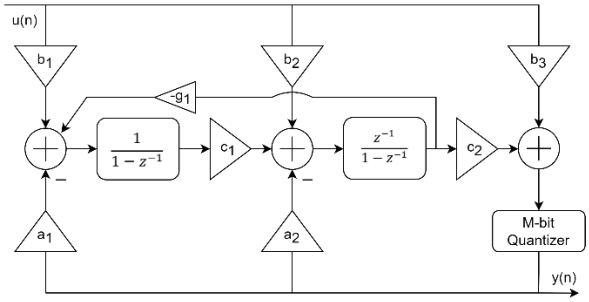


Figure 42 – CRFB2.

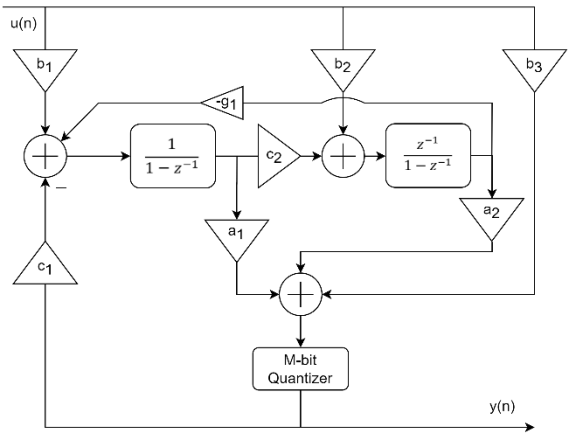


Figure 43 – CRFF2.

Models for the topologies in the Schreier Library that were presented in section 2.10.4 are more complex and the library helps to compute the gain factors that should be used but these values often come as decimal numbers that are not great to use in a digital system and they should be transformed into sums of powers of 2 which can be interpreted in digital systems as right and left shifts. Of course, there is some error attached to this process and the maximum error allowed can be chosen in the models.

As was seen before, this system is supposed to work with 48-bit signed binary data. There are two ways of dealing with signed data, use a mid-rise quantizer where the number of levels is even or use a mid-thread quantizer where the number of levels is odd. A mid-rise quantizer with smallest possible number of levels would have 2 levels with a step size of 2, for example, -1 and +1, whereas a mid-thread quantizer would have 3 levels with a step size of 1, for example, -1, 0 and +1, this last being denoted as a 1.5-bit quantizer. The model accepts either of the quantizer types.

Another characteristic is the possibility to use a gain block after the input. This may be useful to reduce output clipping in some topologies. Of course, this attenuation would have to be compensated by a gain in the DAC full-scale voltage afterwards.

Some of the python code that was developed is shown below in python code 1 to 4. Examples for the EFM2 and CIFB2 are shown here. The rest of the code like auxiliary functions and other modulators can be found in the appendix.

Python code 1 - Digital Sigma-Delta Modulator class

```
import deltasigma as ds
import numpy as np
import aux_sdm as aux

class DSDM:
    def __init__(self, top, osr, order, qbits, in_bits, qtype = 1,
                 H_inf = 1.5, max_err = 0.01, gain = 1):
        #oversampling ratio
        self.osr = osr
        #modulator's order
        self.order = order
        #maximum error when converting decimals to sums of powers of 2
        self.max_err = max_err
        #quantization bits
        self.qbits = qbits
        #infinity gain of NTF
        self.H_inf = H_inf
        #input word length
        self.in_bits = in_bits
        #modulators topology
        self.top = top
        #quantizer type: 1 - mid-rise, 2 - mid-thread
        self.qtype = qtype
        #input attenuation to prevent clipping
        self.gain = gain
```

Python code 2 - Quantization function

```
def quantize(self, v, s):
    #truncation
    y = v >> s
    #convert ...-3 to -5, -2 to -3, 0 to 1; 2 to 3; 3 to 5... (odd ints.)
    if self.qtype == 1:
        if y == 0:
            y = 1
        else:
            y = int(y + y/abs(y) * (abs(y)-1))
    #limiter
    MAX = abs((2**(self.qbits-1))-1)
    if self.qbits == 1:
        MAX = 1
    if y > MAX:
        y = MAX
    if y < -MAX:
        y = -MAX
    return y
```

Python code 3 - 2nd order Error Feedback Digital Sigma-Delta Modulator model

```
def EFM2(self, un = []):
    #number of samples
    ns = un.size
    #output array
    yn = np.zeros(ns).astype(np.int64)
    #error array
    e = np.ones(ns).astype(np.int64)
    #quantizer specs
    b = self.in_bits + 1 #expected quantizer input word length
    s = b - self.qbits #right shift
    #simulation over time
    for n in range(2, ns):
        #input summation
        v = int(un[n] + 2 * e[n-1] - e[n-2])
        #quantization
        y = self.quantize(v, s)
        #compute error feedback
        e[n] = v - (y << s-1)
        #output
        yn[n] = y
    return yn
```

Python code 4 - N[th] order Cascade of Integrators with distributed Feedback Digital Sigma-Delta Modulator model

```
def CIFB(self, un):
    #create NTF for CIFB
    NTF = ds.synthesizeNTF(order = self.order, osr = self.osr,
                           opt=1, H_inf = self.H_inf)
    form = 'CIFB'
    a, g, b, c = ds.realizeNTF(NTF, form)
    ABCD = ds.stuffABCD(a, g, b, c, form)
    #scale coefficients for right output value
    ABCDscaled_out = ds.scaleABCD(ABCD, nlev = 2**(self.qbits))
    ABCDscaled = ABCDscaled_out[0]
    umax = ABCDscaled_out[1]
    a, g, b, c = ds.mapABCD(ABCDscaled, form)
    #coefficients rounding to shifts
    A, B, C, G, a, b, c, g = self.coef2shifts(a, b, c, g)
    #compute new NTF after rounding
    ABCDs = ds.stuffABCD(a, g, b, c, form)
    NTF, STF = ds.calculateTF(ABCDs)
    #number of samples
    ns = un.size
    #output array
    yn = np.zeros(ns).astype(np.int64)
    #intermidiate states array
    xx = [0 for i in range(self.order)]
    #quantizer specs
    b = self.in_bits #expected quantizer input word length
    s = b - self.qbits #right shift
    #simulation over time
    for n in range(0, ns):
        u = int(un[n])
        #output summation
        vA = sum([c[0] * (xx[-1] >> abs(c[1])) if c[1] < 0 else
                  c[0] * (xx[-1] << c[1]) for c in C[-1]])
        vB = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                  b[0] * (u << b[1]) for b in B[-1]])
        v = vA + vB
        #quantization
        y = self.quantize(v, s)
        yn[n] = y
        #output feedback
        yf = y << s
        #intermidiate summations
        for j in range(self.order - 1):
            q = 2 + j
            x2A = sum([c[0] * (xx[-q] >> abs(c[1])) if c[1] < 0 else
                       c[0] * (xx[-q] << c[1]) for c in C[-q]])
            x2B = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                       b[0] * (u << b[1]) for b in B[-q]])
            x2C = sum([a[0] * (yf >> abs(a[1])) if a[1] < 0 else
                       a[0] * (yf << a[1]) for a in A[-(q-1)]])
            xx[-(q-1)] += x2A + x2B - x2C
        #input summation
        x1A = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                   b[0] * (u << b[1]) for b in B[0]])
        x1B = sum([a[0] * (yf >> abs(a[1])) if a[1] < 0 else
                   a[0] * (yf << a[1]) for a in A[0]])
        xx[0] += x1A - x1B
        xx[-2] -= sum([g[0] * (xx[-1] >> abs(g[1])) if g[1] < 0 else
                       g[0] * (xx[-1] << g[1]) for g in G[0]])
    return yn
```

4.4 Current-Steering DAC Model

The CSDAC model has the following characteristics:

1. Resolution and segmentation level variability – it is possible to simulate the DAC with any total number of bits and any segmentation level.
2. Current source mismatch – each current source has an individual random current following a gaussian distribution with some mean that satisfies the resolution and output swing relationship, and a relative standard deviation that is one of the input parameters on the simulation.
3. Differential output – the output is calculated by the difference of two output branches. The numerical value of some current source can be added to one of two output currents summations, depending on its control signal.
4. Current source finite output impedance – when converting the current numerical value to a voltage numerical value, it is multiplied by an equivalent output resistance that is calculated as in equation 3.11. The individual output impedance of each current source, which is considered to be the same for every one of them, is divided by the number of current sources connected to that node at that point in time. While a constant impedance value is generally used in these types of simulation, it is more accurate to use an output impedance that varies with the output voltage as was seen in section 3.2.5. For that, a quick electrical simulation of a current source can be used to get its output impedance as a function of the output voltage for a certain frequency. These simulations values can them be used in the model instead of the constant value.
5. Output settling time – the capacitance at the output and the resistance at that node create an exponential response with a settling time that can be modelled by a simple RC filter at the output.
6. Thermal and flicker noise – the model considers MOSFET thermal and flicker noise and resistor thermal noise.

The python code that models the CSDAC can be seen below in python code 5 and 6.

Python code 5 – Differential Current Steering DAC model

```
import numpy as np
from bitstring import Bits
import aux_sdm as aux
from tqdm import tqdm
from numpy.random import normal
from scipy.signal import filtfilt, butter

def diff_CSDAC(un, nbits, seg, CL, RL, VFS, omega, Rout, fs, time, gm,
               fn0, fnc, nppclk = 1, test = False, DAC_noise = False):
    #binary and unary weighted bits
    bwbits = int((1-seg)*nbits)
    tcbits = int(np.ceil(seg*nbits))
    #number of levels in the converter
    N = 2**nbits
    #LSB current value
    Iref = VFS/(2*RL*(N-1))
    #get Zout characteristic from csv file
    try:
        nofile = False
        Rout_array, V_array = Zout('cadence\\Zout_DC11.csv')
        Rout_array = Rout_array/np.abs(Rout_array).min() * Rout
    except FileNotFoundError:
        RoutP = Rout
        RoutN = Rout
        nofile = True
    #noise corner frequency and flicker noise upper bound
    _4KT = 4*1.380649e-23*300
    fnc = 1e5
    #compute flicker noise coefficient
    kf =  _4KT/gm * fnc
    #flicker noise lower bound
    fn0 = 1e1
    #Signal characteristics
    if test:
        nppclk = 1
        un = np.arange(0, 2**(nbits), 1)
        time = np.arange(0, 2**nbits/fs, 1/fs)
    ns = un.size
    y = np.zeros(ns * nppclk)
    yf = np.zeros(ns * nppclk)
    VoutP = np.zeros(ns)
    VoutN = np.zeros(ns)
    IPbw = 0
    INbw = 0
    IPtc = 0
    INtc = 0
    #Create termometer code current sources
    tc_array = np.array([np.sum(np.array([Iref * (1 + normal(0, omega))
                                          for j in range(2**bwbits)]))
                         for i in range(int(2**tcbits-1))])
    tc_arrayP = np.insert(np.cumsum(tc_array), 0, 0)
    tc_arrayN = np.insert(np.cumsum(np.flip(tc_array)), 0, 0)
    #Create binary weighted current sources
    bw_array = np.flip(np.array([sum(np.array([Iref * (1 + normal(0, omega))
                                               for i in range(int(2**j))]))
                                 for j in range(bwbits)]))
    #Compute flicker noise array if wanted
    if DAC_noise:
        fnoise_array = aux.computePinkNoise(fn0, fnc, kf, gm, time)
```

*Continues in the next page.

Python code 6 - Differential Current Steering DAC model (continuation)

```
    #Simulate DAC
    for n, u in enumerate(tqdm(un)):
        u_ = 2**nbits-u-1
        #computing binary weighted current
        if bwbits != 0:
            bbw = (Bits(uint = u, length = nbits).bin)[tcbits:]
            ubw = int(bbw, 2)
            bintbw = np.array([int(b) for b in bbw])
            IPbw = bintbw.dot(bw_array)
            not_bintbw = np.array([int(not b) for b in bintbw])
            INbw = not_bintbw.dot(bw_array)
        else:
            ubw = IPbw = 0
        ubw_ = 2**bwbits - 1 - ubw
        #computing thermometer coded current
        utc = (u >> bwbits)
        IPtc = tc_arrayP[utc]
        utc_ = len(tc_arrayP) - utc - 1
        INtc = tc_arrayN[utc_]
        #computing output resistance
        if not nofile:
            if n == 0:
                RoutP = Rout_array[0]
                RoutN = Rout_array[-1]
            else:
                RoutP = Rout_array[np.abs((V_array - VoutP[n-1])).argmin()]
                RoutN = Rout_array[np.abs((V_array - VoutN[n-1])).argmin()]
        #RL in parallel with u or u_ current source output impedances
        if u != 0:
            ReqP = abs(aux.shunt(RL, RoutP/(u)))
        else:
            ReqP = RL
        if u_ != 0:
            ReqN = abs(aux.shunt(RL, RoutN/(u_)))
        else:
            ReqN = RL
        #compute DAC thermal and flicker noise if wanted
        if DAC_noise != 0:
            #current source thermal noise
            tnoise = np.random.normal(0, np.sqrt(
                2 * ns * _4KT * ((N-1) * gm * RL**2 + 2 * RL)))
            #current source flicker noise
            fnoise = fnoise_array[n] * np.sqrt(N-1) * RL
            #total output noise
            noise = tnoise + fnoise
        else:
            noise = 0
        #linear output current
        IoutP = IPtc + IPbw
        IoutN = INtc + INbw
        #computing output voltage
        y[n * nppclk: (n+1) * nppclk] += IoutP * ReqP - IoutN * ReqN + noise
        #computing output test voltage
        VoutP[n] = IoutP * ReqP
        VoutN[n] = IoutN * ReqN
    #add settling time to output voltage by RC circuit
    tau = RL*CL
    yf = y * (1-np.exp(-time/tau))
    return yf
```

4.5 Reconstruction Filter Model

First and second order gm-C filters were modelled by its transfer function. If higher order filters are needed a cascade of first and second order filters may be considered. The transconductance cells were considered to all have the same value of transconductance and to have the following non-idealities:

- Non-zero output conductance go.
- A high frequency zero at $1/\tau$ with the transconductance gain being equal to:

$$gm \cdot (1 - s \cdot \tau). \tag{4.8}$$

- Non-zero node parasitic capacitance C_p.
- Noise in the filter in analyzed at the end of sections 4.5.1 and 4.5.2 and is considered in the model.

4.5.1 First-Order gm-C Filter

A first order gm-C filter can be realized as in figure 44 where the gm blocks are transconductance amplifiers and C is a capacitor.

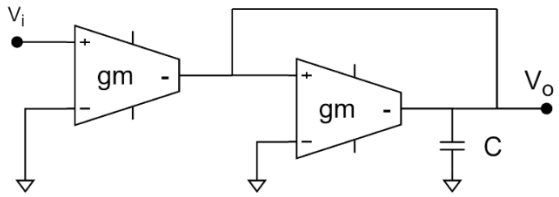


Figure 44 – First order gm-C filter.

Analyzing the output node Vo with KCL and considering the non-idealities mentioned previously, the transfer function in equation 4.9 can be obtained. The total conductance and capacitance at the output node are considered.

$$H(s) = \frac{-gm(1 - s\tau)}{s\left(C + C_{p_{total}} - gm\tau\right) + go_{total} + gm} \tag{4.9}$$

In ideal conditions the transfer function would be:

$$H(s) = \frac{-gm}{sC + gm}. \tag{4.10}$$

The ideal cutoff frequency is given by:

$$\omega_o = \frac{gm}{C}, \tag{4.11}$$

and the real cutoff frequency is given by:

$$\omega_o = \frac{gm + go_{total}}{C + C_{p_{total}} - gm\tau}. \tag{4.12}$$

From the previous three equations it is clear that the non-idealities can have a big impact on the behaviour of the filter if they are not minimized.

Another factor to consider in the design of the gm-C filter is the noise introduced in the system by the gm cells. The noise analysis can be done considering that the noise sources are uncorrelated and by grounding the input and adding a random current source at the output as depicted in figure 45.

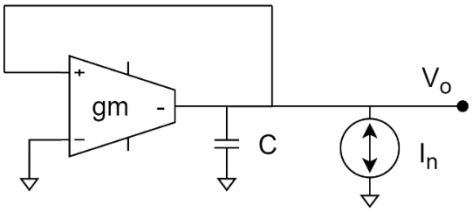


Figure 45 – Noise analysis of first order gm-C filter.

The output referred noise power spectrum density can be calculated by:

$$S_{n_{out}}{}^2(s) = \left|\frac{V_o}{I_n}\right|^2 \cdot \bar{I_n}^2, \tag{4.13}$$

$$\bar{I_n}^2 = 4K_B T\gamma gm. \tag{4.14}$$

In this case, the first term of the multiplication in equation 4.13 is just the output impedance so the output referred noise power spectrum density is given by equation 4.16 after using KCL on the output node of figure 45 which yields

$$I_n + V_o \cdot sC + V_o \cdot gm = 0 \leftrightarrow \frac{V_o}{I_n} = \frac{1}{sC + gm}. \tag{4.15}$$

Taking the absolute value squared of the output impedance:

$$\left|\frac{V_o}{I_n}\right|^2 = \left|\frac{1}{j\omega C + gm}\right|^2 = \left|\frac{gm - j\omega C}{\omega^2 C^2 + gm^2}\right|^2 =$$

$$= \left(\sqrt{\left(\frac{gm}{\omega^2 C^2 + gm^2} \right)^2 + \left(\frac{-\omega C}{\omega^2 C^2 + gm^2} \right)^2} \right)^2 = \frac{\omega^2 C^2 + gm^2}{(\omega^2 C^2 + gm^2)^2} =$$

$$= \frac{1}{\omega^2 C^2 + gm^2} = \frac{\frac{1}{gm^2}}{1 + \left(f \cdot \frac{2\pi C}{gm} \right)^2}.$$

By equation 4.13:

$$S_{n_{out}}{}^2(f) = \frac{\frac{4K_B T\gamma}{gm}}{1 + \left(f \cdot \frac{2\pi C}{gm} \right)^2}. \tag{4.16}$$

The total output noise power or the noise variance considering an ideal transfer function can be derived to be:

$$V_n{}^2 = \int_{-\infty}^{\infty} S_{n_{out}}(f)\, df = \frac{K_B T\gamma}{C}, \tag{4.17}$$

so, the noise power is inversely proportional to the value of the capacitor C.

In this application what is important is the in-band noise output noise PSD. It can be approximated to be equation 4.16 evaluated at f = 0 yielding:

$$S_{n\_inband} \approx S_{n_{out}}(0) = \frac{4K_B T\gamma}{gm} = \frac{4K_B T\gamma}{\omega_o C}. \tag{4.18}$$

4.5.2 Second-Order gm-C Filter

A general biquad section has a second order low pass transfer function of the type:

$$H(s) = \frac{K}{s^2 + \frac{\omega_0}{Q} s + \omega_0{}^2}, \tag{4.19}$$

where $K/\omega_0{}^2$ is the gain, ω_0 is the cutoff frequency and Q is the quality factor of the filter. A gm-C implementation of 4.19 is depicted in figure 46.

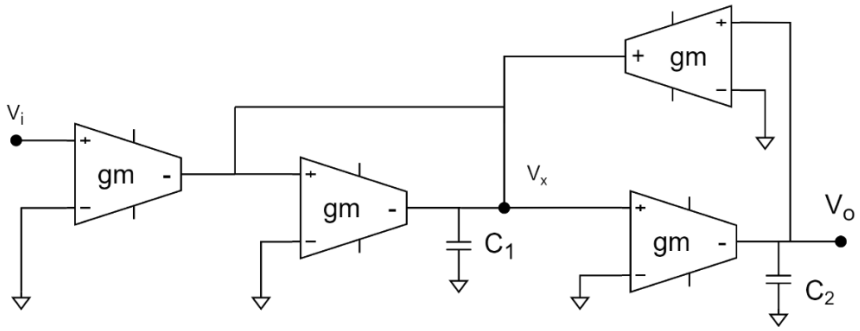


Figure 46 – Second order gm-C filter.

Analyzing nodes Vx and Vo with KCL we get equations 4.20. The total conductance and capacitance at both nodes are considered.

$$\begin{cases} V_x \cdot (sC_1 + sC_{px} + g_{ox} + gm') - V_o \cdot gm' + V_i \cdot gm' = 0 \\ V_o \cdot (sC_2 + sC_{po} + g_{oo}) + V_x \cdot gm' = 0 \\ gm' = gm \cdot (1 - s \cdot \tau) \end{cases} \tag{4.20}$$

Ideally the TF would be:

$$H(s) = \frac{gm^2}{C_1 C_2 s^2 + gm \cdot C_2 s + gm^2}. \tag{4.21}$$

Comparing coefficients in 4.19 with those in 4.21 the ideal cutoff frequency, gain and quality factor can be calculated as in equations 4.22.

$$\begin{cases} \omega_o = \dfrac{gm}{\sqrt{C_1 C_2}} \\ Q = \sqrt{\dfrac{C_1}{C_2}} \\ H(0) = 1 \end{cases} \tag{4.22}$$

With the following simplifications:

$$\begin{cases} D = gm \cdot \tau \\ g_{ox} = 3g_o \\ g_{oo} = g_o \\ C_1' = C_1 + C_{px} \\ C_2' = C_2 + C_{po} \end{cases}, \tag{4.23}$$

the real TF is given by:

$$H(s) = \frac{\dfrac{s^2 D^2 - sgm2D + gm^2}{(D^2 - C_2' D + C_1' C_2')}}{s^2 + s\dfrac{(3C_2' + C_1' - D)g_o + (C_2' - 2D)gm}{(D^2 - C_2' D + C_1' C_2')} + \dfrac{gm^2 + 3{g_o}^2 + gmg_o}{(D^2 - C_2' D + C_1' C_2')}}. \tag{4.24}$$

As was seen for the first order filter, the non-idealities can cause big changes in the behaviour of the filter, changing the ideal cutoff frequency, quality factor, DC gain and introducing zeros in the transfer function.

A noise analysis can be done in a similar way to equation 4.16 and with the method used in the previous section. These calculations would be very difficult to perform exactly. Author [25] arrives at equation 4.25 for the variance of the thermal noise in the second order band-pass gm-C filter.

$$V_n{}^2 = \frac{4K_B T\gamma Q}{C_1} \tag{4.25}$$

Because for this application only the in-band noise PSD is important, there is no need for such calculations. It can be computed in a similar way as was done in the previous section but with two noise sources, one for each node, and using the superposition theorem.

$$\frac{V_o}{I_{nx}} = \frac{-gm}{C_1C_2s^2 + gm \cdot C_2 s + gm^2} \quad \text{and} \quad \frac{V_o}{I_{no}} = \frac{gm + sC_1}{C_1C_2s^2 + gm \cdot C_2 s + gm^2}$$

Setting s = 0, taking the absolute value squared, multiplying by the current PSD in equation 4.14, and adding everything, the in-band output noise PSD is approximately given by:

$$S_{n_{out\_inband}} \approx S_{n_{out}}(0) = \frac{8K_B T\gamma}{gm} = \frac{8K_B T\gamma}{\omega_o C_2 Q} \tag{4.26}$$

The python code used to model the filters is shown below in python code 7 and 8.

Python code 7 - gm-C reconstruction filter model

```
import scipy.signal as sig
import numpy as np
import aux_sdm as aux

def rec_filt(x, N, wo, Q, gm, fs, go, Cp, tau, time, fn0, fnc, filter_noise=False):
    ns = x.size
    _4KT = 4*1.380649e-23*300
    fnoise1 = np.zeros(ns)
    fnoise2 = np.zeros(ns)
    #first order filter for odd order filter implementations
    if N%2 != 0:
        #filter transfer function and frequency response
        C1 = gm/wo
        Cp1 = np.random.normal(0, Cp/3) * C1
        go1 = np.random.normal(0, go/3) * gm
        num1 = [gm * tau, -gm]
        den1 = [C1 + Cp1 - gm*tau, gm + go1]
        numz1, denz1 = sig.bilinear(num1, den1, fs)
        f1, h1 = sig.freqz(numz1, denz1, worN = np.arange(0, fs, fs/ns), fs = fs)
        h = h1
        f = f1
```

*Continues in the next page.

Python code 8 - gm-C reconstruction filter model (continuation)

```
        #filter noise
        if filter_noise:
            noise1 = _4KT/gm
            max_noise = noise1
            kf1 =  _4KT/gm * fnc
            fnoise1 = aux.computePinkNoise(fn0, fnc, kf1, gm, time)
            noise = np.random.normal(0, np.sqrt(2 * ns * max_noise), ns)
            noise += fnoise1/gm
    #second order filter for orders bigger than 1
    if N != 1:
        #filter transfer function and frequency response
        C22 = gm/(wo*Q)
        C21 = C22*Q**2
        D = gm*tau
        go2 = np.random.normal(0, go/3)*gm
        C21 *= (1+np.random.normal(0, Cp/3))
        C22 *= (1+np.random.normal(0, Cp/3))
        print("C21 = {:e} F".format(C21))
        print("C22 >= {:e} F".format(C22))
        a2 = D**2/(D**2-C22*D+C21*C22)
        a1 = -gm*2*D/(D**2-C22*D+C21*C22)
        a0 = gm**2/(D**2-C22*D+C21*C22)
        b2 = 1
        b1 = ((3*C22+C21-D)*go2 + gm*(C22-2*D))/(D**2-C22*D+C21*C22)
        b0 = (gm**2 + 3*go2**2 + gm*go2)/(D**2-C22*D+C21*C22)
        num2 = [a2, a1, a0]
        den2 = [b2, b1, b0]
        numz2, denz2 = sig.bilinear(num2, den2, fs)
        f2, h2 = sig.freqz(numz2, denz2, worN = np.arange(0, fs, fs/ns), fs = fs)
        f = f2
        #filter noise
        if filter_noise:
            noise2 = 2 *_4KT/gm
            max_noise = noise2 * int(N/2)
            kf2 =  _4KT/gm * fnc
            fnoise2 = aux.computePinkNoise(fn0, fnc, kf2, gm, time)
    #time domain filtering and noise
    if N%2 == 0:
        #second order cascade
        h = h2**(N/2)
        if filter_noise:
            noise = np.random.normal(0, np.sqrt(2 * ns * max_noise), ns)
            noise += (fnoise2/gm)*(N/2)
        x += noise
        y = sig.lfilter(numz2, denz2, x)
        for n in range(int(N/2)-1):
            y = sig.lfilter(numz2, denz2, y)
    else:
        #odd order
        if N != 1:
            h *= h2**(int(N/2))
            if filter_noise:
                noise += (fnoise2/gm) * int(N/2)
        x += noise
        y = sig.lfilter(numz1, denz1, x)
        for n in range(int(N/2)):
            y = sig.lfilter(numz2, denz2, y)
    return f, h, y, noise
```

4.6 VCO Model and Maximum Input Noise

When a VCO in placed in a PLL loop it can be viewed as a system whose input is the control voltage and the output is the excess phase mentioned in equation 4.3. It can be modelled by the following transfer function:

$$\frac{\theta_{ex}}{V_{in}}(s) = \frac{K_v}{s}, \tag{4.27}$$

so, it operates as an ideal integrator with DC gain K_v [22].

When it comes to noise, the analog part of the system should dominate over the digital part. The main source of noise will be the transconductances in the gm-C filter. Table 2 has the goal phase noise values with which it is possible to calculate the maximum white noise level at the input of the VCO. Using equation 4.4, each phase noise value yields a different noise floor shown in table 3.

Table 3 – Output phase noise conversion to input white noise

Offset	VCO Output Phase noise [dBc/Hz]	VCO Input White Noise [dB/Hz]
30 Hz	-100	-136
100 Hz	-114	-140
1 kHz	-138	-144
10 kHz	-153	-139
100 kHz	-161	-137
1 MHz	-166	-112
10 MHz	-166	-92

There is a value of input white noise power that when converted to VCO output phase noise as shown in figure 31 is always below the specification phase noise curve whose points are in the second column of table 3. This value is in the third column of table 3 for an offset of 1 kHz. This is, of course, considering only white noise. If pink noise that is dependent on frequency like flicker noise is in the system, the noise level may have to be adjusted.

4.7 Noise and Jitter Modelling

For this application there is a need to model thermal and flicker noise. Figure 47 shows an example of a spectrum with both noises. The corner frequency where thermal noise becomes preponderant over flicker noise will be defined as fc.

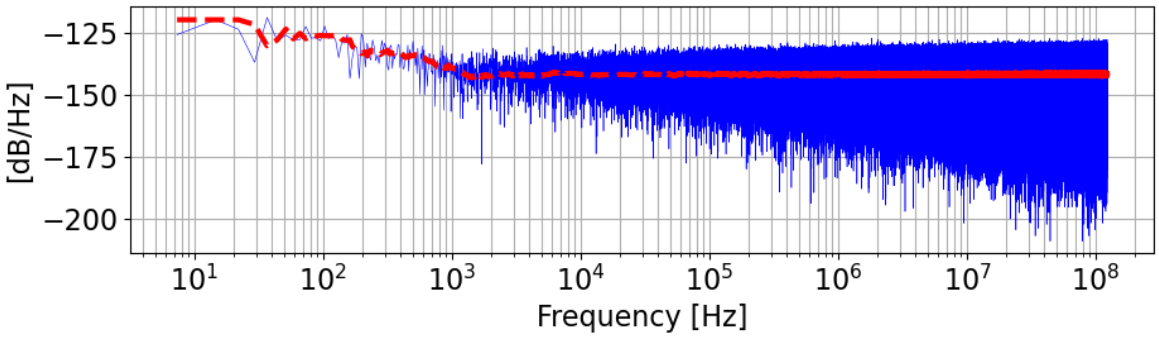


Figure 47 – Example of flicker and thermal noises with a corner frequency of 1 kHz.

The fastest computational way to simulate thermal noise is considering it white noise with some standard deviation and zero mean and generating such a random signal with gaussian distribution and the same number of points as the signal to which the noise is supposed to be summed. In previous sections, PSDs for thermal noise have been derived. From those equations it is possible to calculate the standard deviation by integrating over the frequency spectrum and taking the square root. Integrating white noise, which is constant over all frequencies, is just multiplying by twice the bandwidth of over which it is being integrate. Because the data being handled here is discrete and has a finite number of points, such integrations can be computed by multiplying by the number of points in the signal. The method for computing discrete time domain thermal noise in the Python programing language is shown in equation 4.28, where n_s is the number of samples in the signal, the parameter loc corresponds to the mean of the distribution, and scale is the standard deviation computed as in equation 4.29.

$$V_{th} = numpy.random.normal(loc = 0, scale = \sigma, size = n_s) \tag{4.28}$$

$$\sigma = \sqrt{2 \cdot n_s \cdot S_{th}} \tag{4.29}$$

Contrary to thermal noise, flicker noise is not that easy to model. Its frequency dependency makes it so it cannot be modeled as white gaussian noise. Instead, author [27] proposed a technique that involves summing up a finite fixed number N_f of random gaussian distributed phase φ_i sinusoids with frequencies f_i in a specified band as in equation 4.30.

$$V_f = \sum_{i=1}^{N_f} a_i \cdot \sin(2\pi f_i t + \varphi_i) \tag{4.30}$$

The band limit can be chosen to be the corner frequency f_c where flicker noise becomes less important than thermal noise. So, choosing a band from f_0 to f_c with N_f, the linear spacing between frequencies will be

$$\Delta f = \frac{f_c - f_0}{N_f}. \tag{4.31}$$

Logarithmic division of the bandwidth may also be used and Δf would be the difference between two consecutive frequencies. So, for each frequency interval f_i to $f_i + \Delta f$, the amplitude of each sinusoid is approximated as

$$a_i = \sqrt{2 \int_{f_i}^{f_i + \Delta f} S_f(f)\, df}, \tag{4.32}$$

where $S_f(f)$ is the flicker noise PSD. Considering the PSD in equation 4.33.

$$a_i = \sqrt{2K \ln\left(1 + \frac{\Delta f}{f_i}\right)} \tag{4.33}$$

As conclusion, for modelling flicker noise one can use equations 4.30 and 4.33 to create a time domain flicker noise voltage array. Depending on the number of points of the time signal and the number of points in the flicker bandwidth N_f the results will be more or less accurate and the simulations will take more or less time.

When it comes to jitter, it can be modelled by assuming that it has a gaussian distribution with mean equal to zero and variance that can be computed using equation 4.1. The approach is to create a time signal with the characteristics mentioned before and add to the time array using in simulation. This process is identical to the one in equation 4.28.

4.8 Frequency Response of Simulated Discrete Time Domain Systems

When simulating systems in discrete time domain often the visualization of the data in the frequency spectrum is as important or even more important that the time domain counterpart. For this purpose, Fourier analysis is generally used with the fast Fourier transform (FFT) being the main algorithm to compute the discrete Fourier transform of a time sequence of data.

Most programing languages offer this tool but for an FFT to be effective one must be careful about some aspects. The FFT needs an infinitely long periodic signal but because in practice there are no infinitely long signals it receives a finite periodic signal as input and uses copies of this signal one after another to compose an infinitely long one. This means that the first and last point must be consecutive points for the result of the FFT to be correct. A If this is not satisfied spectral leakage will occur and signal power which was supposed to be concentrated in one single bin of the spectrum will be spread to other adjacent bins. For this not to happen coherent sampling must be used. The steps for coherent sampling are displayed below.

1. Fix a sampling frequency for the system (fs).
2. Choose the number of points in the input signal which will correspond to the number of points also used in the output spectrum (ns). For fast computation, ns which is a power of two should be used so, $ns = 2^k$.
3. The division of fs by ns yields the frequency spacing between bins in the output spectrum (Δf), so for more resolution higher ns should be used at the cost of a slower simulation.
4. For coherent sampling the input signal's frequency must be a multiple of Δf so, $fin = m \cdot \Delta f$. Usually the frequency of the input signal is chosen by setting the number of periods (np) to ns divided by some power of two bigger than 2 itself and then adding a 1 to make np relatively prime with ns. The input frequency is calculated by $fin = np \cdot fs/ns$.

Due to the impulsive response of the modulator, a technique called windowing must be used. This technique is performed by multiplying a time domain signal with zeros at the beginning and at the end and with some shape in between, by the input time signal thereby forcing the input signal of the FFT to start and begin at the same point. This comes at the cost of wider frequency bins and an increase in the power of all of them. How much wider and how much more power the bins will have depends on the order and type of window being used. In this work, the Hanning window will be used as it is the recommended type for sigma-delta modulators output signals. To remove the extra power added per bin by the window the output is divided by the windows first order norm over 2.

Another important characteristic of the FFT spectrum is that it is not the same as power spectral density unless the number of samples matches the sampling frequency and $\Delta f = 1$.

Generally, power in a FFT bin will integrated power over a bandwidth of Δf. If a PSD measurement is needed, Δf must be set to one or alternatively the power of one bin can be divided by Δf to get power over a 1 Hz bandwidth. When measuring phase noise in dBc/Hz, this procedure must be done for correct measurement.

4.9 Simulations, results, and conclusions

In this section simulation results are presented. The approach to be used is as follows: as it is predicted that noise at low and intermediate frequencies is dominated by the analog part, first, it is going to be defined that noise floor and the respective filter gm specification to achieve it. Also, noise from the filter is independent of the cutoff frequency as it will be defined by the choice of capacitors with the transconductance fixed due to noise requirements and that noise will most likely dominate that of the DAC. After defining this noise floor from the analog part, sigma-delta modulators are tested to see which one yields the best quantization noise spectrum which will dominate at high frequencies. These modulators are simulated with mid-rise quantizers and step size of 2. The cutoff frequency and order of the filter will be chosen so that this quantization noise is attenuated to become lower than the specification limit. At the end the systems that satisfy the specifications are compared to determine which should be better in terms of cost, area, and power consumption.

The gm-C filter transconductance value to achieve the required noise level can be determined by using equations 4.18 and 4.26 and with the values of phase noise converted to VCXO input white noise in table 3. Because of flicker noise, the noise levels determined in table 3 that have an offset below the corner noise must be decreased 10 dB per decade because flicker noise will have a descending slope of -10 dB per decade from DC to the corner frequency. It is then essential to define the corner frequency. Reference [25] reports a gm cell with noise corner frequency of 300 Hz. In this work a f_c of 1 kHz will be considered to have some margin of error. By the reasoning explained before the transconductance gm used in the gm-C filter of N^{th} order is determined to be:

$$gm > N \cdot 22\ \mu S. \tag{4.34}$$

The expected analog noise spectrum is shown in figure 48 and its phase noise conversion is in figure 49.

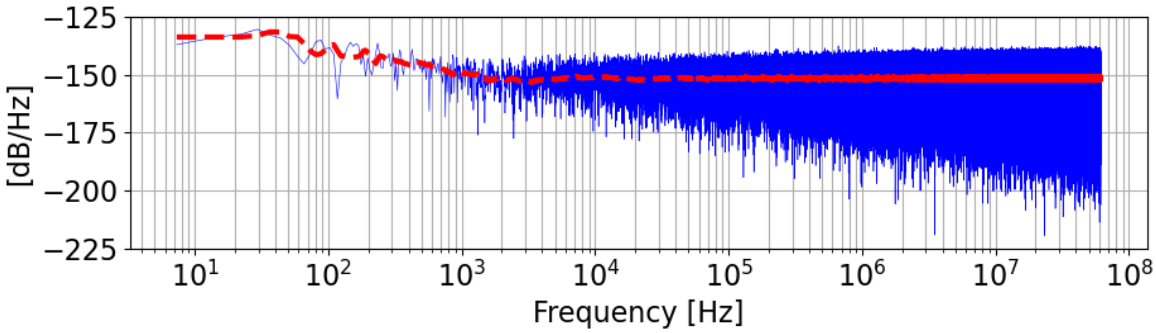


Figure 48 – Noise from analog reconstruction filter.

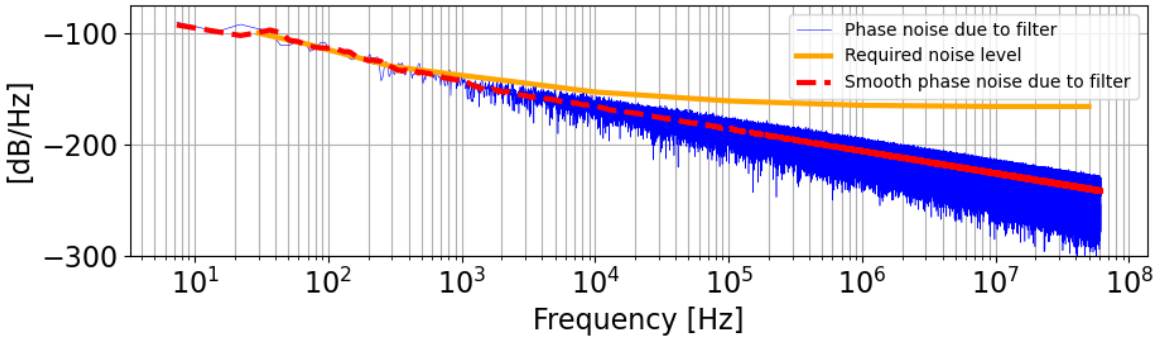


Figure 49 – Phase noise due to analog reconstruction filter.

As can be seen in figure 49, the high frequency phase noise due to the filter is way below the required level, meaning that there is some room for the high frequency modulated quantization noise from the digital sigma-delta modulator that precedes the reconstruction filter. Table 4 shows the different systems that were simulated with sinusoidal waves with a frequency calculated as in equation 4.35 in order to perform coherent sampling. With this input frequency the OSR is approximately $2^{16} = 65536$.

$$f_{in} = F_s \cdot \frac{periods}{ns} = 122.88\ MHz \cdot \frac{2^7 + 1}{2^{24}} \approx 945\ Hz \tag{4.35}$$

One characteristic that has not been mentioned is the slew rate needed for the DAC and filter. The slew rate can be defined as the maximum rate of change of voltage over time as in equation 4.36.

$$SR = \max\left(\left|\frac{dv(t)}{dt}\right|\right) \tag{4.36}$$

The worst case happens for single bit quantization but, in this case, because the cutoff frequency of the filter is so much lower than the sampling frequency of the modulator, the filter

will behave practically like an integrator that will produce the average value of the modulators output. If cutoff frequencies close to the Nyquist frequency are considered. Than slew rate problems may arise. Also, 1-bit truncation has a poorly defined gain factor which might lead to instability and clipping of the modulator for high swing inputs. For these reasons, the systems were simulated with 1 and 5 output bits. The second case is represented with a " * " as in B1*. The 1-bit modulators were tested with full scale and 50% of full-scale sine waves to see the effect of clipping on the spectrum. The 5-bit modulators and the MASH were tested only with full-scale sine waves. The results of VCXO input noise generated by the systems in table 4 plus the analog noise of the filter before filtering are shown in figures 60 to 67. They have been smoothened and the input signal removed to facilitate visualization. The transconductance cell considered has a gm of 80 μS which is bigger than the value stated in equation 4.34 and leaves a 6 dB margin from the phase noise requirement. The SFDR specification of 100 dBc at the output of the VCXO can be converted to the input with equation 4.6 and its dependent on the frequency at which the spur occurs. The spectrums of all modulators in which SFDR can be measured are shown in figures 89 to 116 in the annexes. The systems were grouped into four categories that can be seen on the left-hand side of table 4. The saturation that affects some modulators can be overcome by adding a small gain factor at the input. This gain factor would then be compensated in the analog DAC full-scale voltage.

Table 4 – Modulators simulated.

Category	**#**	**Modulator**	
		Order	**Topology**
Basic ΣΔ modulators	**B1**	1st	OFM
	B2	1st	EFM
	B3	2nd	OFM
	B4	2nd	EFM
MASH ΣΔ modulators	**M1**	2nd	EFM MASH
	M2	3rd	EFM MASH
Complex ΣΔ modulators	**C1**	2nd	CIFB
	C2	2nd	CRFB
	C3	2nd	CIFF
	C4	2nd	CRFF

The modulators block diagrams are repeated here for clarity in figures 50 to 59 with their simulated maximum word length in the intermediate registers. All the bit values are for signed words with 1 bit reserved as sign bit. For simplicity, the CIFB, CRFB, CIFF and CRFF word lengths and coefficients are only shown for the 1-bit quantization and the information for MASH2 is not shown. However, the tool allows to see all this information after each simulation.

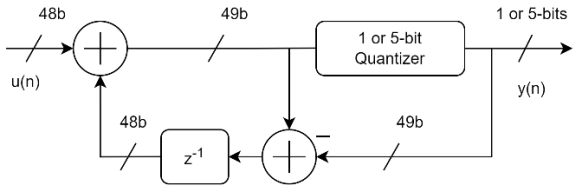


Figure 50 - EFM1 with word lengths.

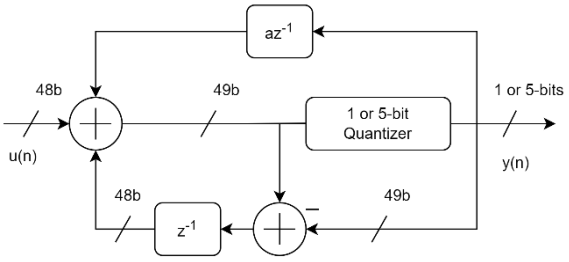


Figure 51 - HKEFM1 with word lengths.

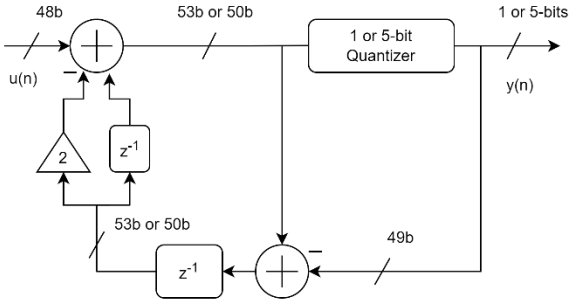


Figure 52 - EFM2 with word lengths.

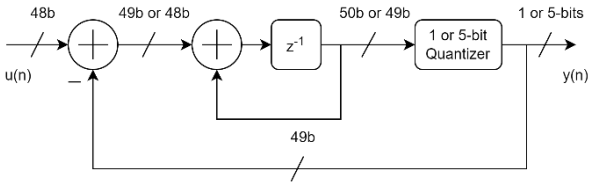


Figure 53 - OFM1 with word lengths.

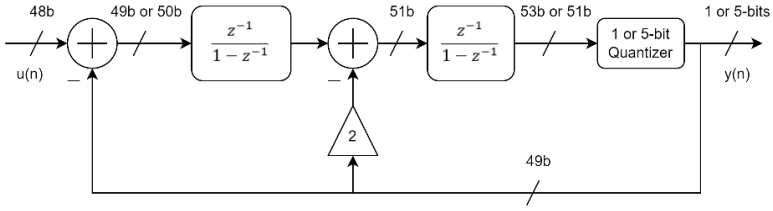


Figure 54 - OFM2 with word lengths.

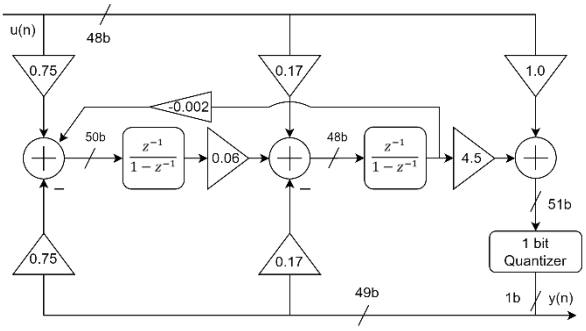


Figure 55 - CIFB with word lengths and coefficients.

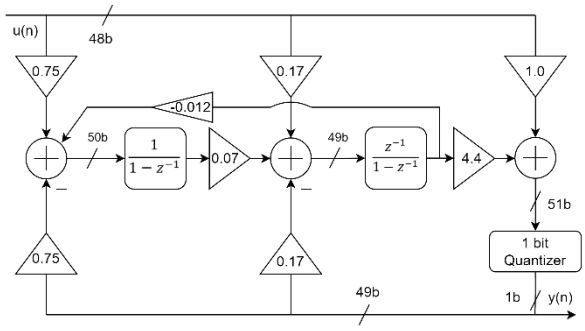


Figure 56 - CRFB with word lengths and coefficients.

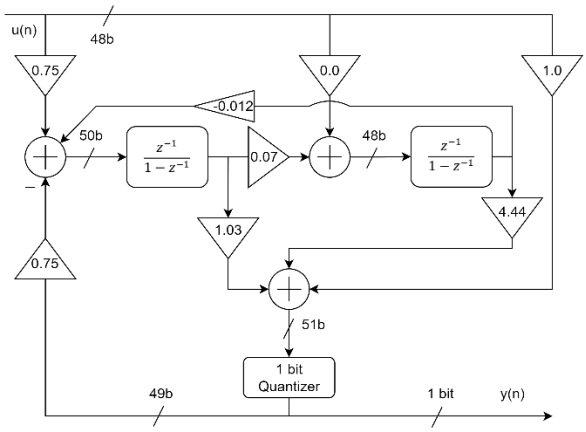


Figure 57 - CIFF with word lengths and coefficients.

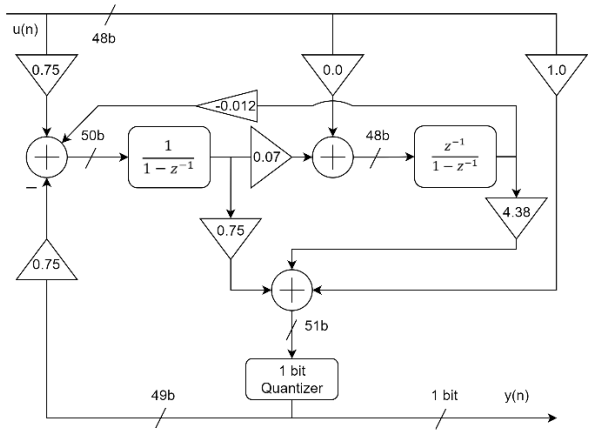


Figure 58 - CRFF with word lengths and coefficients.

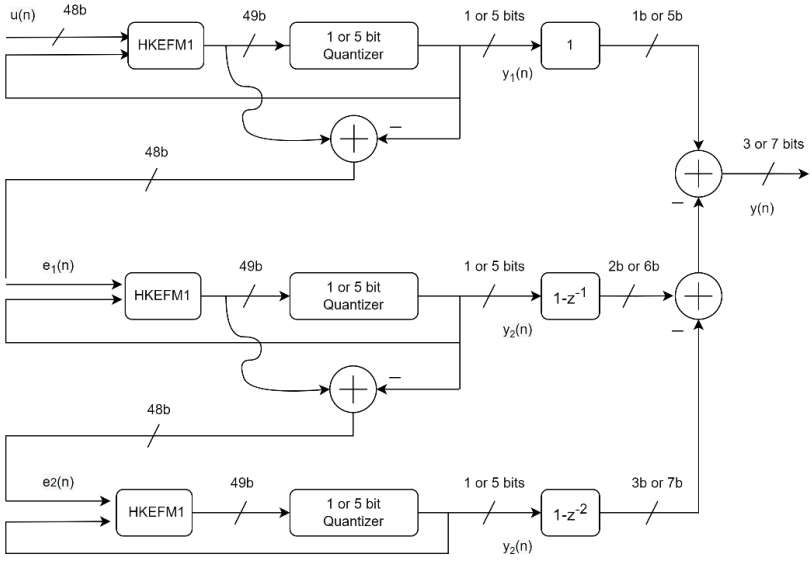


Figure 59 - MASH3 with word lengths and coefficients.

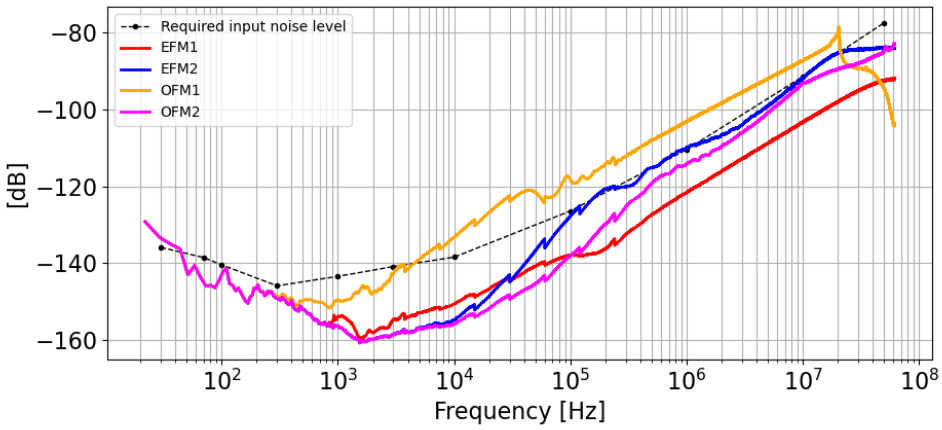


Figure 60 – VCXO input noise before filtering for B1, B2, B3 and B4 with full-scale input.

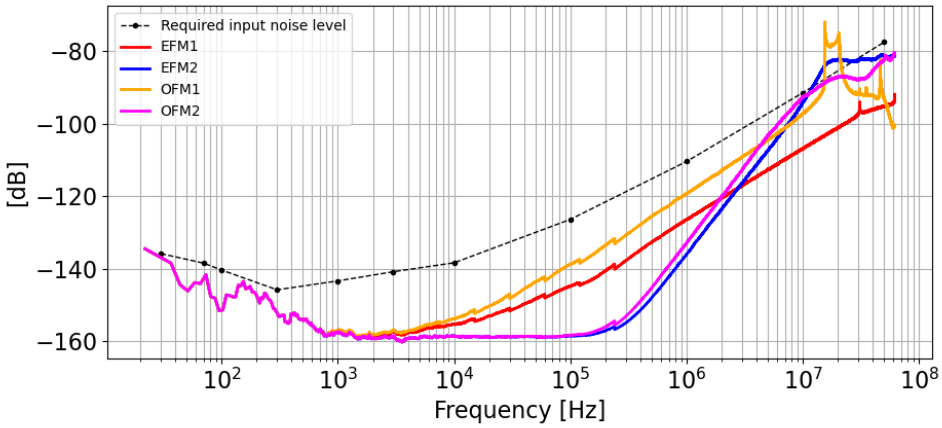


Figure 61 – VCXO input noise before filtering for B1, B2, B3 and B4 with half-scale input.

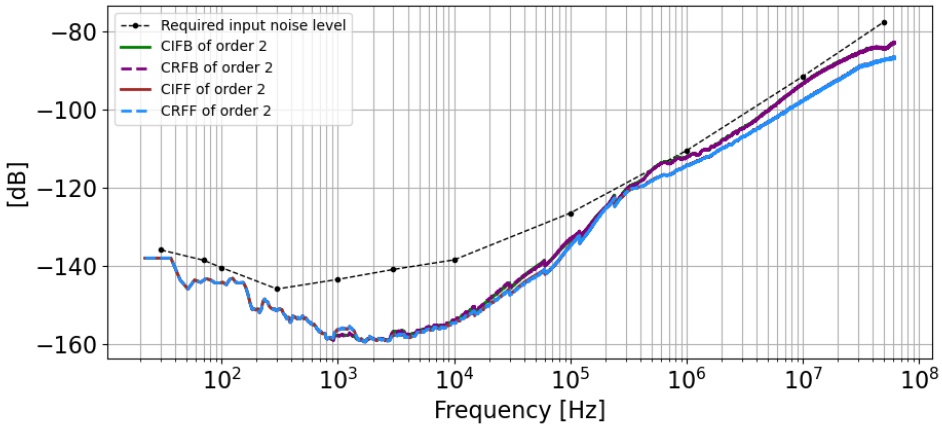


Figure 62 – VCXO input noise before filtering for C1, C2, C3 and C4 with full-scale input.

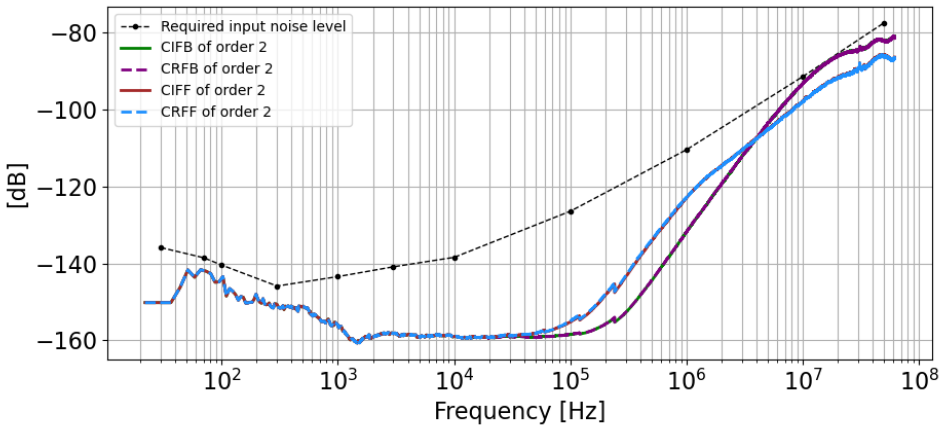


Figure 63 – VCXO input noise before filtering for C1, C2, C3 and C4 with half-scale input.

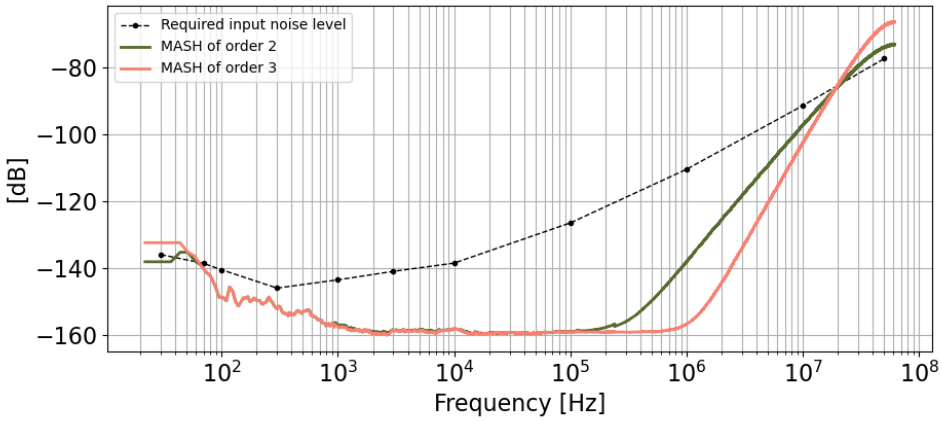


Figure 64 – VCXO input noise before filtering for M1 and M2 with full-scale input.

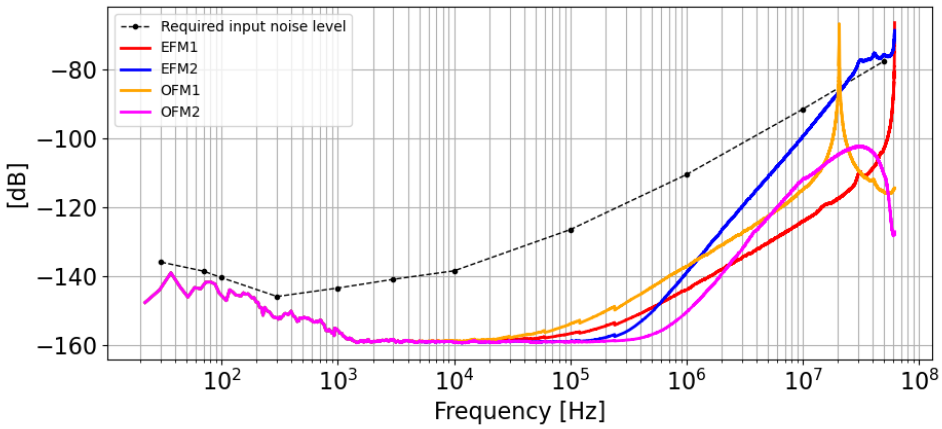


Figure 65 – VCXO input noise before filtering for B1*, B2*, B3* and B4* with full-scale input.

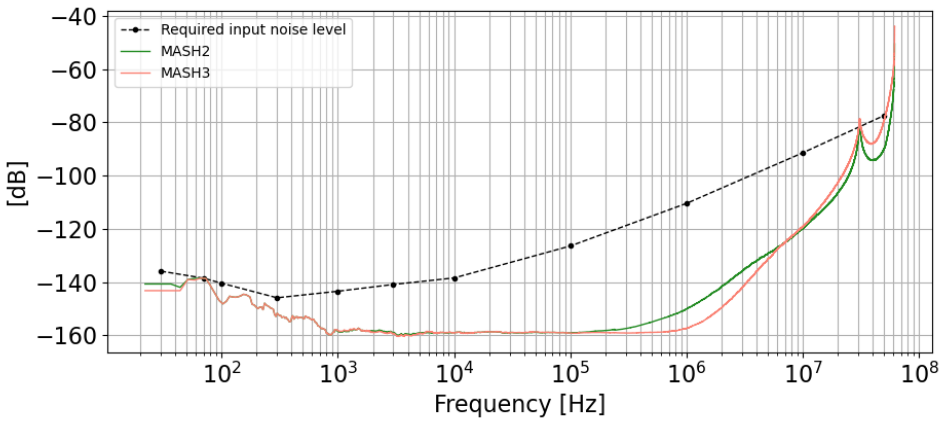


Figure 66 – VCXO input noise before filtering for M1* and M2* with full-scale input.

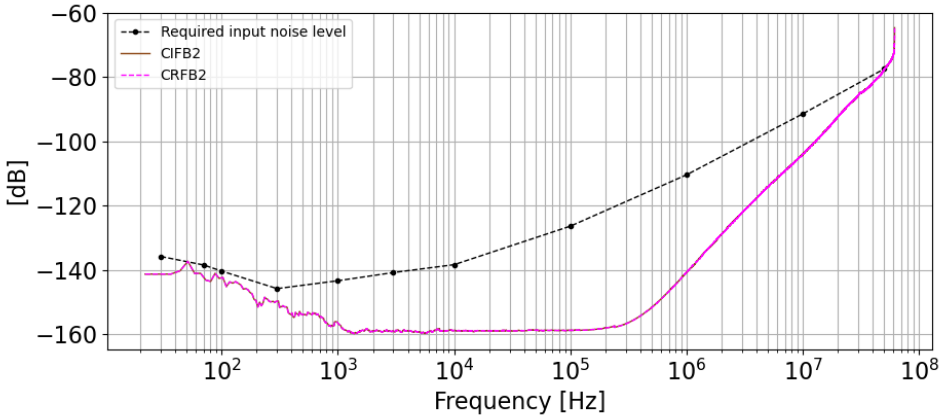


Figure 67 – VCXO input noise before filtering for C1* and C2* with full-scale input.

Modulators with more than 1 output bit need to be simulated with a current steering DAC to find out what is the maximum relative current standard deviation that keeps the systems within the specified limits. The non-linearities caused by this mismatch with appear as unwanted spurs in the spectrum and also may raise the quantization noise floor from the DAC itself as seem in the example of figure 68 for a EFM2 with 5-bit quantization and a CSDAC with different relative standard deviation values. After some repeated simulations (50 to 100 iterations for several values of relative standard deviation), the summary results are in table 5. More iterations should have been done but due to time constraints it was not possible, so these results are only first approximations and further testing should be done to certify them. They show that the required values of relative current standard deviation are very low for some systems and will lead to large devices by equation 3.9. DEM techniques may be need if

extremely large devices are to be avoided. Regarding the SFDR specification, the minimum current source output impedance to achieve it was simulated and is also presented in table 5.

Table 5 – CSDAC current mismatch and minimum output impedance requirements to meet specifications.

#	Resolution	$\frac{\sigma(I)}{I}$	Rout [kΩ]	#	Resolution	$\frac{\sigma(I)}{I}$	Rout [kΩ]
M1	2b	0.0003	10	**B4***	5b	0.001	75
M2	3b	0.0003	20	**M1***	6b	0.001	150
B1*	5b	0.0001	75	**M2***	7b	0.001	300
B2*	5b	0.0001	75	**C1***	5b	0.002	75
B3*	5b	0.001	75	**C2***	5b	0.002	75

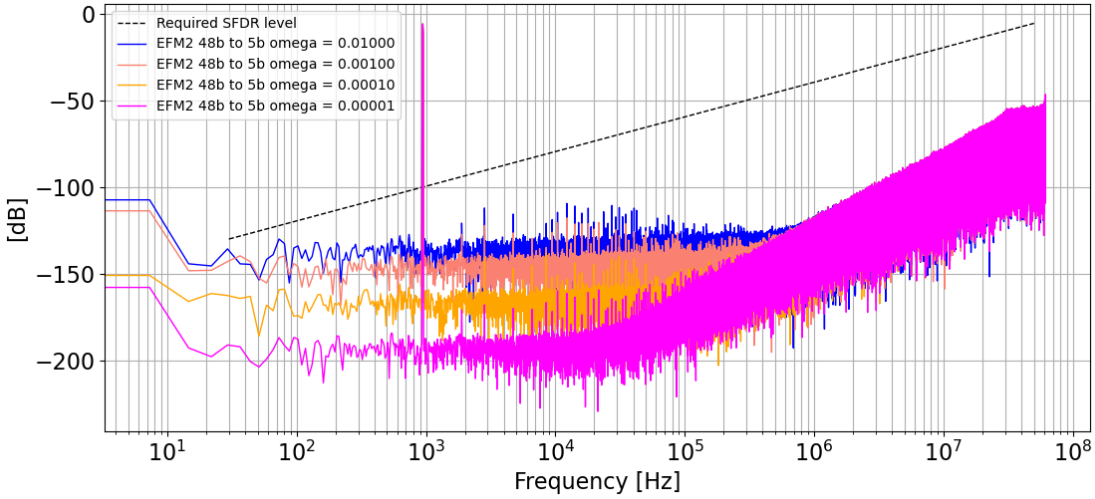


Figure 68 – Examples of spectrum for a 48b to 5b EFM2 with mismatch in the current sources of the DAC.

In order to compare each system with each other, table 6 was built with some characteristics. A few conclusions can be taken:

1. All systems pass the noise and SFDR requirements after the analog reconstruction filtering is applied.
2. 1-bit output modulators obviously have much simpler DACs that use less area and power, but most of them show clipping of the output when full-scale inputs are used, and because the slew rate needed at the input of the filter is higher, the savings in power will be spent on that block. Overall, there is a saving in area and complexity of the analog part at the cost of full-scale clipping, harmonic distortion, and perhaps more power consumption by the filter.

3. EFM architectures show all-round better performance than OFM, specially the more basic ones. More complex structures in the C category show good performance but the improvement is not worth the extra digital logic that is used.
4. MASH modulators show the best performance of all at the cost of some additional logic, but the increase in the number of output bits may be costly.
5. The cutoff frequencies and orders shown in table 6 are just enough to keep the high frequency quantization noise below the limit and they all yield capacitors of 12 pico-Farads or less. Lower cutoff frequencies can be used, and higher order can be chosen.

Table 6 – Summary of the simulation results.

#	# of z^{-1}, × and + in modulator	Noise spec. [Y/N]	SFDR spec. [Y/N]	Full-scale clipping [Y/N]	Harmonic distortion [0, +, ++, +++]	DAC Resolution, VFS, levels	Filter cutoff frequency, order
B1	3	Y	Y	Y	+++	1b, 1V, 2	1 MHz, 1st
B2	3	Y	Y	N	+++	1b, 1V, 2	60 MHz, 1st
B3	5	Y	Y	Y	+	1b, 1V, 2	1 MHz, 1st
B4	7	Y	Y	Y	+	1b, 1V, 2	3 MHz, 1st
M1	11	Y	Y	N	0	2b, 3V, 4	10 MHz, 1st
M2	16	Y	Y	N	0	3b, 7V, 8	10 MHz, 2nd
C1	12	Y	Y	Y	+	1b, 1V, 2	10 MHz, 1st
C2	13	Y	Y	Y	+	1b, 1V, 2	10 MHz, 1st
C3	12	Y	Y	Y	++	1b, 1.5V, 2	10 MHz, 1st
C4	13	Y	Y	Y	++	1b, 1V, 2	10 MHz, 1st
B1*	3	Y	Y	N	++	5b, 2V, 32	10 MHz, 1st
B2*	3	Y	Y	N	++	5b, 2V, 32	60 MHz, 1st
B3*	5	Y	Y	N	++	5b, 2V, 32	5 MHz, 1st
B4*	7	Y	Y	N	+	5b, 2V, 32	10 MHz, 1st
M1*	11	Y	Y	N	+	6b, 3.9V, 64	10 MHz, 1st
M2*	16	Y	Y	N	+	7b, 7.8V, 128	10 MHz, 1st
C1*	12	Y	Y	N	++	5b, 1.9V, 32	60 MHz, 1st
C2*	13	Y	Y	N	++	5b, 1.9V, 32	60 MHz, 1st

As a final conclusion, there does not seem to be a need for a second order modulator as first order show an adequate performance, but for the price of a just a few more digital logic gates and a slightly lower cutoff frequency, a second order modulator is recommended for an improved quantization noise level and less harmonic distortion. When it comes to quantization bits, 1 bit seems to be enough. Multi-bit quantization seems to be only advisable if the slew rate requirement for 1-bit quantization is too hard to achieve, as mismatch constraints may present a difficult challenge in multi-bit DACs.

As an example, the following system was simulated: a 1-bit EFM2 with an input attenuation of $1 - 2^{-3} - 2^{-5} = 0.84375$ inside the modulator to remove clipping of the modulator. The filter would be a first order gm-C with gm equal to 80 uS and a capacitor of 4.25 pF, yielding a cutoff frequency of 3 MHz. The DAC would be a simple highly linear 1-bit DAC that could be implemented with current steering, switched capacitors or any other architecture. The block diagram, the signals viewed in the time and frequency domain, and the resulting VCXO phase noise of such a system are shown in figures 69 to 75. Looking at the time domain signal that drives the VCXO in figure 70, it can be said that, even after filtering, it is still corrupted by a lot of noise, but it can be argued that the VCXO itself and the remaining of the PLL system will act as a low pass filter removing that excessive high frequency noise. Either way, the specification that was given is met by the system as can be seen in figure 75.

This is just a suggestion of configuration that meets the requirements, others may be looked into, and further simulation and analysis may be done to find the optimal approach. Also, because there are some systems that have a performance much superior to the required, perhaps there is an opportunity for power saving by reduction of the clock frequency. Of course, because the 122.88 MHz clock is already in the PLL, that reduction would have to be implemented by other circuit that would consume power, but the tradeoff may be worth it, and it is a possibility to study in future work.

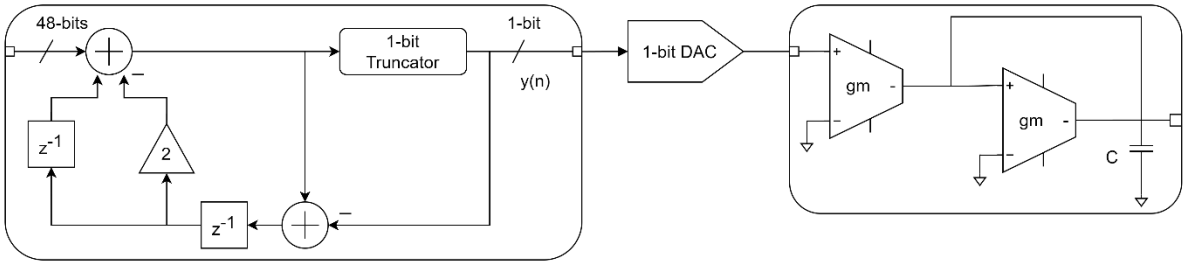


Figure 69 – 48b to 1b EFM2 with gm-C reconstruction filter block diagram.

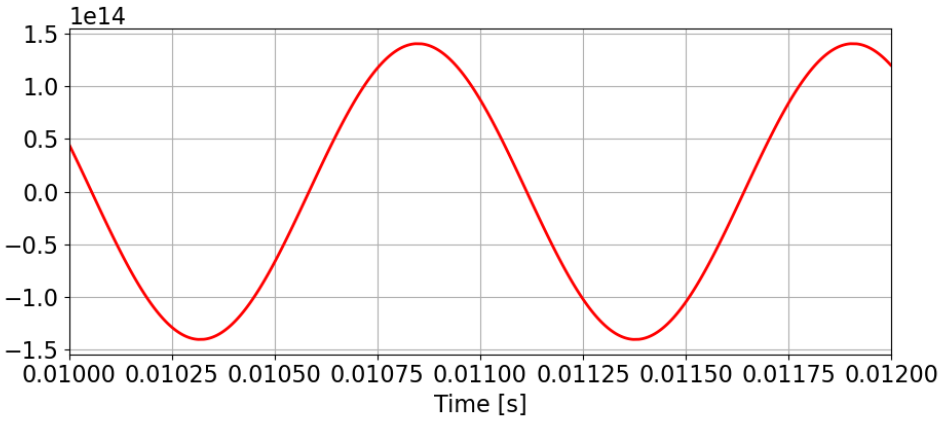


Figure 70 – System's 48-bit signed full-scale sine wave time domain input.

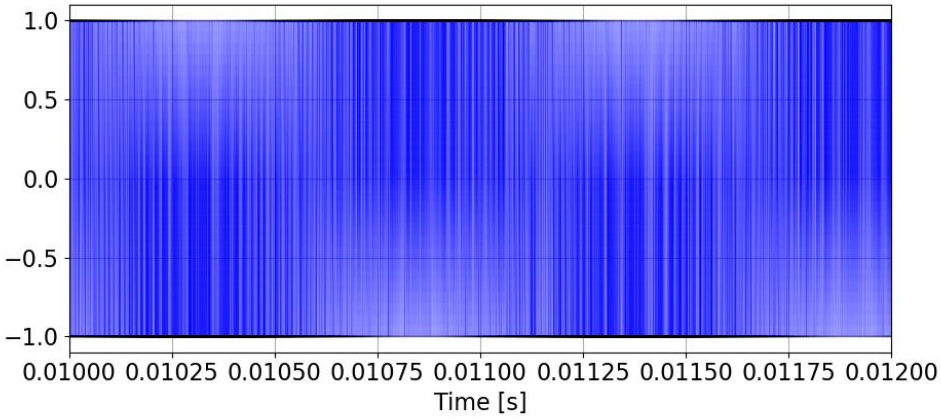


Figure 71 – 48-bit to 1-bit EFM2 time domain output.

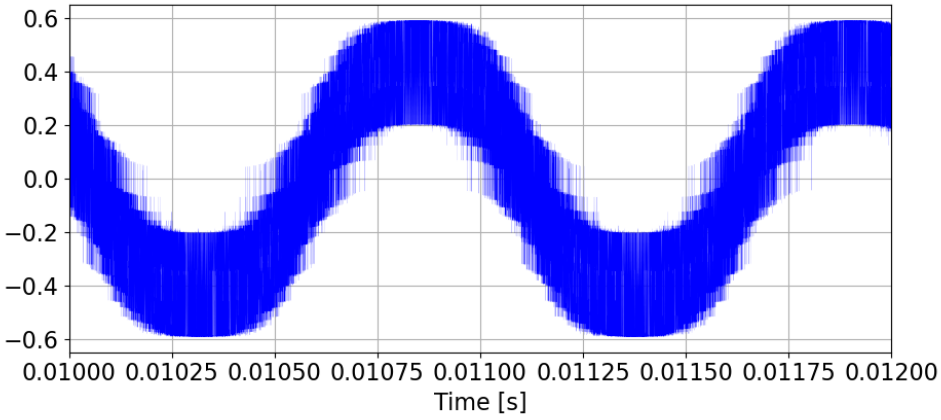


Figure 72 – 48-bit to 1-bit EFM2 time domain output filtered by 1st order at 3 MHz.

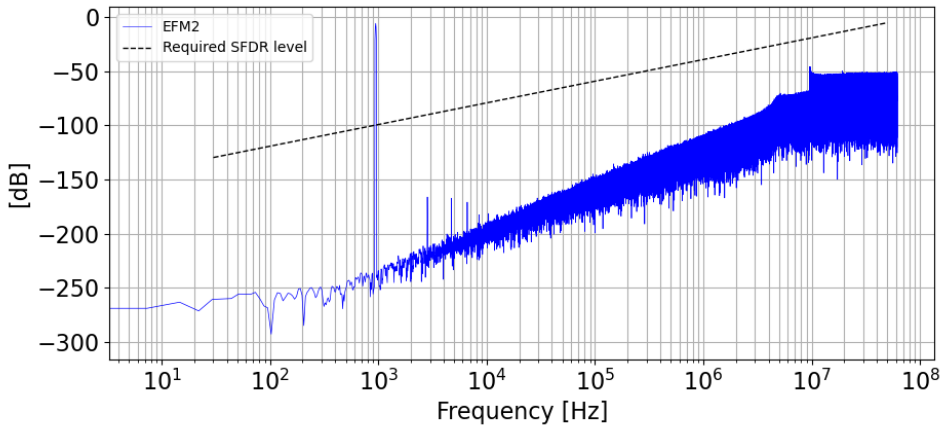


Figure 73 – 48-bit to 1-bit EFM2 frequency domain output.

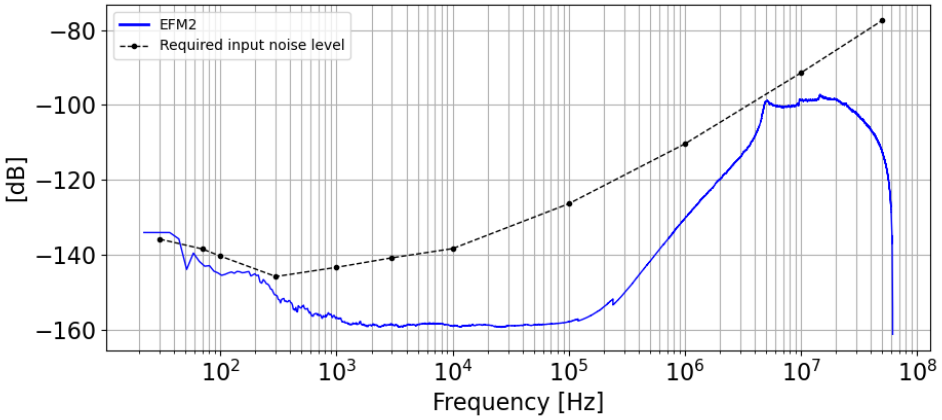


Figure 74 – 48-bit to 1-bit EFM2 smoothed frequency domain output noise filtered by 1st order at 3 MHz.

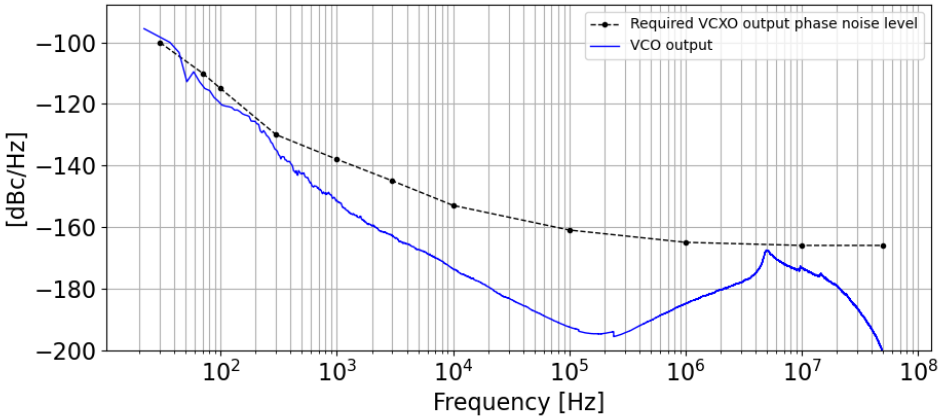


Figure 75 – VCXO output phase noise due to 48-bit to 1-bit EFM2 filtered by 1st order at 3 MHz.

5

CURRENT STEERING DAC - DESIGN AND SIMULATION

In this chapter the design of a current steering digital to analog converter is presented. The model for the CSDAC presented in the previous chapter is used to further understand the requirements to accomplish the specifications in table 7.

Table 7 – DAC specifications.

Specification	**Value**
Sampling frequency - Fs	122.88 MHz
Resolution - M	10 bits → N = 1023
Load conversion resistance R_L	50 Ω
Output Swing	Differential – 0.5 V Full Scale
INL	< 0.5 LSB
DNL	< 0.5 LSB
SFDR	> 60 dBc
Load capacitance C_L	10 pF
Technology	UMC 130 nm
Power supply	1.2 V

5.1 Current Cell Design

To achieve a 0.5 V full scale in a differential output, the LSB current can be calculated as follows:

$$I_{LSB} = \frac{VFS}{2NR_L} \approx 4.8828\,\mu A. \tag{5.1}$$

As was shown by equation 3.8 in section 3.2.4, the relative standard deviation of the current source current can be calculated for a specific yield. Generally, a yield of 99.7% is used and, for a maximum INL of half an LSB:

$$\frac{\sigma(I)}{I} \leq \frac{0.5}{inv_{norm_{(-x,+x)}}\left(0.5 + \frac{0.997}{2}\right) \cdot \sqrt{2^{10}}} \approx 0.00527. \tag{5.2}$$

In a differential output configuration, the SFDR is limited by the HD3. Using equation 3.13, one can determine the minimum value of the output impedance to achieve the desired SFDR of 60 dBc.

$$|Z_{out}| > \frac{N \cdot R_L \cdot 10^{\frac{60}{40}}}{4} \approx 404\ K\Omega \tag{5.3}$$

Of course, this is considering that the output impedance is constant over the time with the output voltage swing that changes the state of the transistors and that there are no other non-linearities in the circuit like transistor mismatch. Also, by equation 3.14, the maximum effect this output impedance can have on the INL is approximately 0.0089 LSB, way below the 0.5 LSB limit.

The cascode configuration in figure 26 a) was chosen for the current cell to be able to achieve the required output impedance and to improve mirroring and the dynamic performance of the converter. Equation 5.2 shows that the relative standard deviation has to be less than 0.00527. By the Pelgrom model and equation 3.9 and using the mismatch values for the UMC 130 nm process we get the system of equations with 5.4.1 and 5.4.2.

$$\begin{cases} W \cdot L \geq \left(\frac{4{A_{V_{TH}}}^2}{(V_{GS} - V_{TH})^2} + {A_\beta}^2\right) \Big/ \left(\frac{\sigma(I)}{I}\right)^2 & (5.4.1) \\ I_{LSB} = \frac{1}{2}\mu_p C_{ox}\frac{W}{L}(V_{GS} - V_{TH})^2 \leftrightarrow W = \frac{2I_{LSB}L}{\mu_p C_{ox}(V_{GS} - V_{TH})^2} & (5.4.2) \end{cases}$$

The only free variable in this system of equations is $(V_{GS}-V_{TH})^2$, the overdrive voltage squared V_{OV}^2. There are two important considerations to have in mind when choosing this value: a higher value minimizes the effect of A_{VTH} and therefore minimizes the total area. On the other hand, for a higher value it is more difficult to maintain all devices in the saturation region. It is important to note that for a full-scale output voltage of 0.5 V, the swing at each output node is at 0 to 0.25V, which leaves 0.95 V to maintain the three devices in the saturation region with some safety margin. The biggest problem encountered when doing this sizing is that the CN node voltage is fixed by the V_{GS} of the switch transistor so the minimum V_{DS} available for the switch to remain in saturation can be calculated to be:

$$V_{dsat} \approx V_{GS} - V_{th}, \\ V_{DS} > V_{GS} - V_{th} \leftrightarrow V_{swing} > -V_{cont} - V_{th} \leftrightarrow V_{cont} > V_{swing} - V_{th}. \quad (5.5)$$

The absolute value of the voltages was considered in these equations. It can be seen that, for instance, for a swing of 0.5 V resulting in a 1V full scale, and with the threshold voltage approximately equal to 0.33 V, the control voltage would have to be larger than 0.167 V, otherwise the switch will work in triode and saturation over the output voltage swing resulting in distortion. There are two possible solutions for this problem: use an especial driver that increases the on voltage for the switch or reduce the output swing. As seen before, in this work a swing of 0.25 V was chosen which is enough to keep all transistors in saturation with some safety margin using full scale control voltages from 0 to Vdd. This is the simpler approach as it does not require changing the control voltages but has the disadvantage of reducing the output swing and increasing the capacitive feedthrough from the control voltage to the output when compared to reduce swing control voltages.

Setting the overdrive voltage of the current source device to 0.3 V and solving the system of equations in 5.4, the sizing for this device is shown in table 8. The cascode device and the switches are sized to have minimum channel length and V_{OV} that always keeps then in saturation. Their sizes are also in table 8.

Table 8 – Sizing of the current cell.

Device	W [µm]	L [µm]	Vov [V]
Mcs	5.400	5.000	0.300
Mcas	0.450	0.120	0.160
Msw	0.450	0.120	0.160

This sizing yields a current source whose current statistical distribution is shown in figure 76 and was obtained through 1000 Monte Carlo simulations in Cadence Spectre. The mean value was measured to be 4.89 uA with a standard deviation of 22.74 nA resulting in a relative standard deviation of 0.00465 which is lower than the 0.00527 that is needed to obtain an INL lower than half LSB.

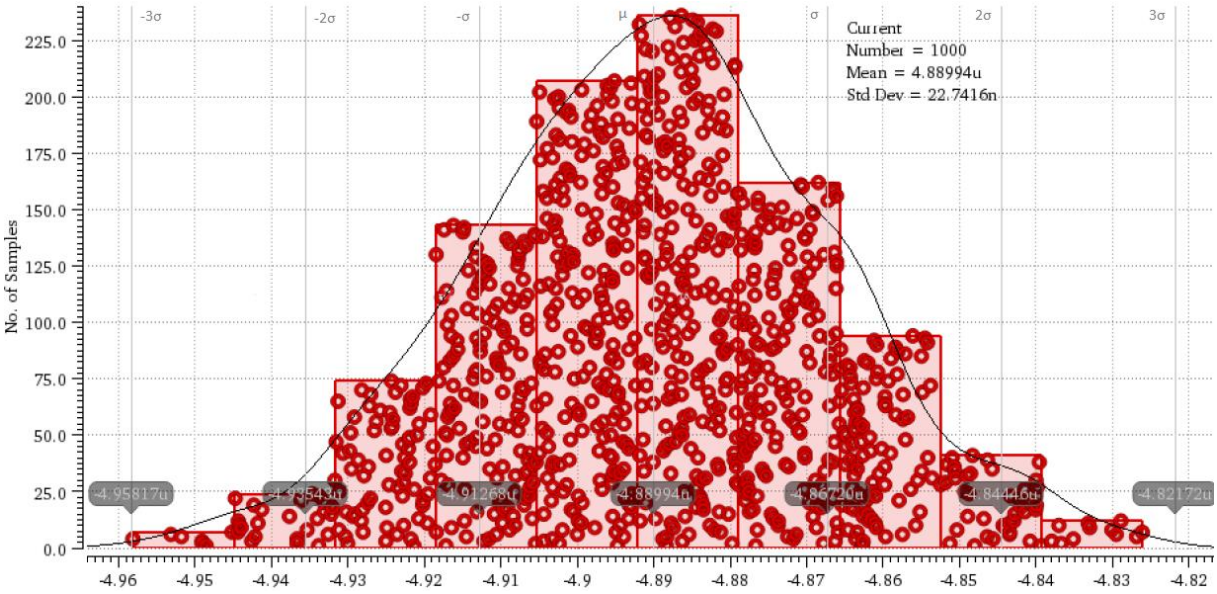


Figure 76 – Current source statistical distribution.

Corner simulations were done, and the results are shown in table 9. The single ended output voltage was fixed at maximum value. The setup for the corners can be found in table 13 in section A1 of the appendix. The worst corner was measured to be number six with the cascode device deep into triode region and an estimated output impedance at Nyquist frequency of 3 MΩ, which is still very much above the minimum theoretical value calculated in equation 5.3.

Table 9 – Current source corner simulations.

Corner # (Setup in A1 of the Appendix)	Current [uA]	Saturation Margin [mV]			Zout @ DC [MΩ]	Zout @ 60 MHz [MΩ]
		CS	CAS	SW		
0	4.890	94	129	104	206	23
1	4.886	95	132	98	155	20
2	4.904	102	32	-42	54	5
3	4.893	115	337	116	269	26
4	4.910	124	238	-21	176	12
5	4.868	46	7	219	38	9
6	4.872	39	-97	76	14	3
7	4.889	71	209	240	164	27
8	4.899	82	105	97	219	25
9	4.884	80	105	134	121	19
10	4.889	87	5	-8	51	6
11	4.894	101	308	153	230	27
12	4.907	111	209	14	210	18
13	4.876	58	41	188	67	13
14	4.886	58	-59	45	25	4
15	4.891	81	244	209	190	27
16	4.902	91	143	67	236	23

To further confirm this assumption, the impedance value for the worst corner and the measured standard deviation (3 MΩ and 0.00465) were used in the DAC high-level model at Nyquist frequency and the measurements from the 1000 iterations are shown below in figures 77 and 78. An example of spectrum is shown in figure 79. The maximum DNL and INL values were simulated to be 0.115 LSB and 0.311 LSB respectively, and the minimum SNDR and SFDR were simulated to be 61.62 dB and 72.45 dBc yielding a minimum ENOB of 9.94 bits.

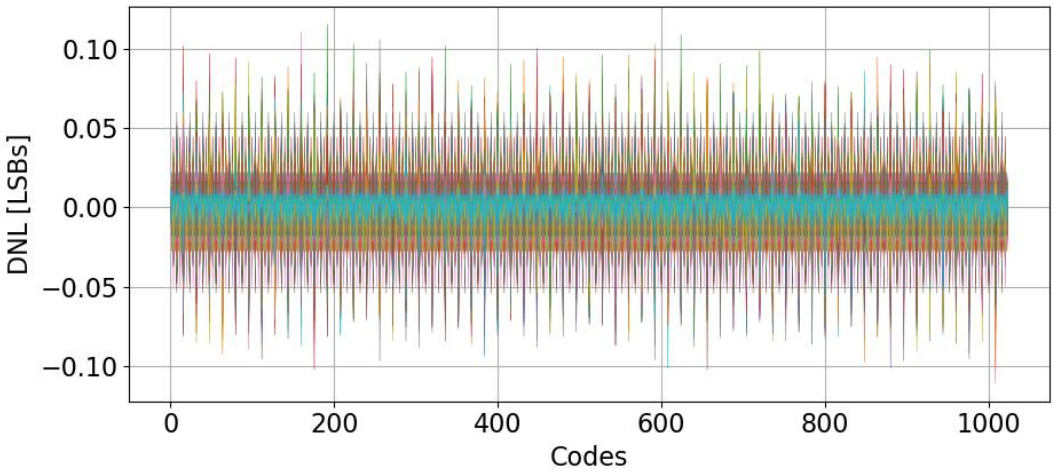


Figure 77 – DNL from 1000 runs of the high-level model.

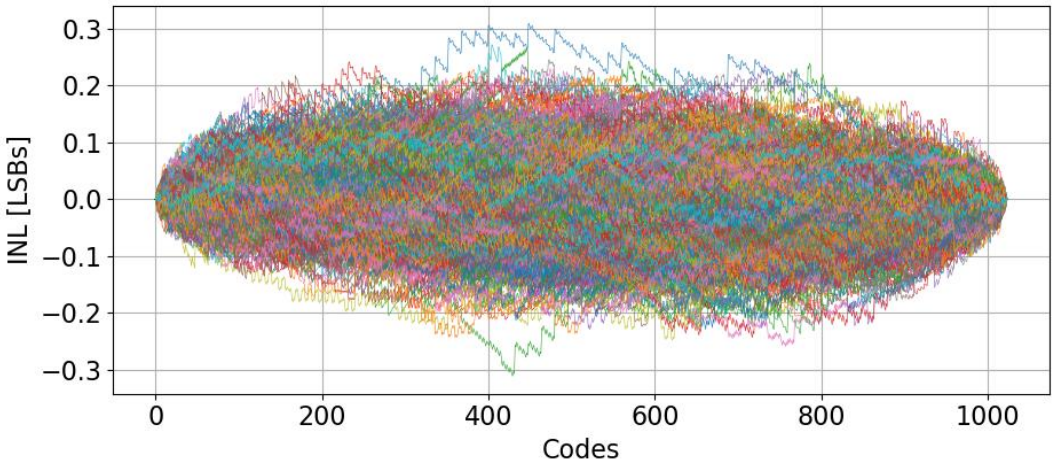


Figure 78 – INL from the 1000 runs of the high-level model.

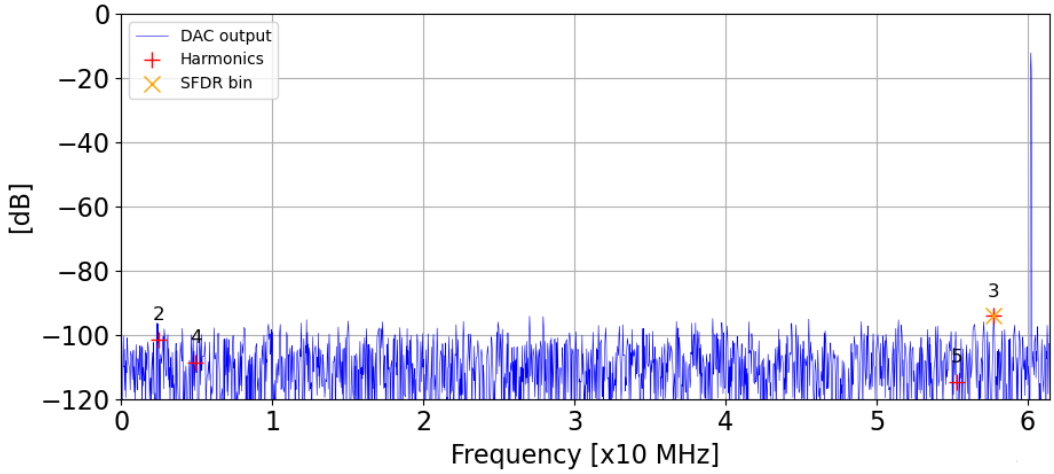


Figure 79 – Example of spectrum from the high-level model.

5.2 Latch and Switch Driver

A part of the current source that has not been discussed yet though is the driver for the switches which, as mentioned in section 3.2.3, needs to guarantee synchronization, and lower the crossing point between the raising and fall edges in order to keep both switches on when a transition is happening. Author [28] proposes the structures in figure 80 and 81 to achieve synchronization and lower crossing point respectively. The first block is composed of a latch controlled by the master clock and inverters that isolate the latch from adjacent parasitic capacitances and generate the needed opposite phase signals to drive the switch driver. This second block lowers the crossing point by generating a signal whose fall time is faster than rise time. This structure was adopted in this work with the sizing shown in table 10. The crossing point value, rise and fall time are shown in table 11 and the crossing signals for the typical corner is shown in figure 82. All inverters were sized with the width of the NMOS equal to 0.450 μm, width of the PMOS being double that of the NMOS and minimum length of 0.120 μm for all devices.

Table 10 – Switch driver sizing.

Device	W [μm]	L [μm]
M_A	0.200	0.120
M_B	0.200	0.120
M_C	0.600	0.120

Table 11 – Latch and switch driver measurements.

Measurement	Typical Corner	Worst Corner
Crossing point	0.344 V	0.407 V @ #12
Rise time	236 ps	412 ps @ #6
Fall time	41 ps	67 ps @ #6

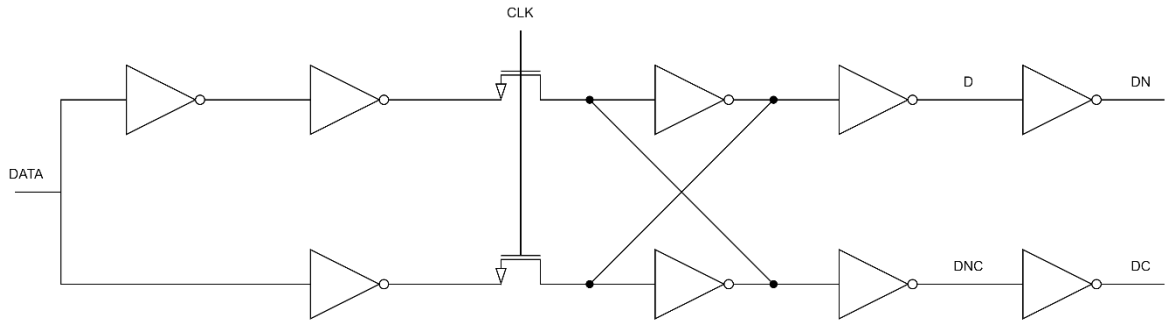


Figure 80 – Latch.

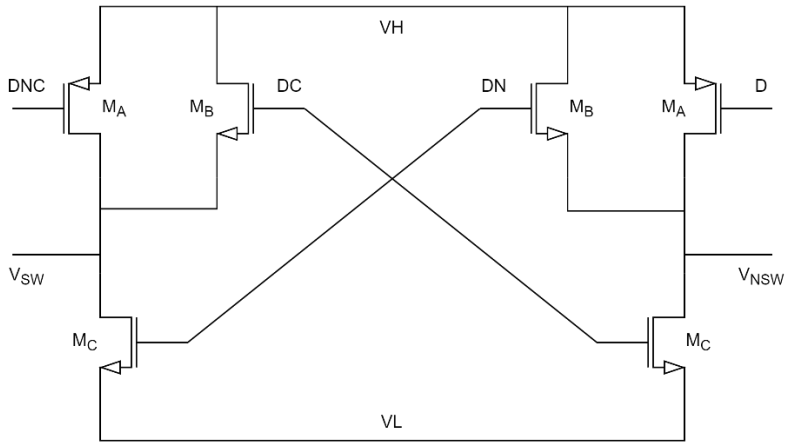


Figure 81 – Switch driver.

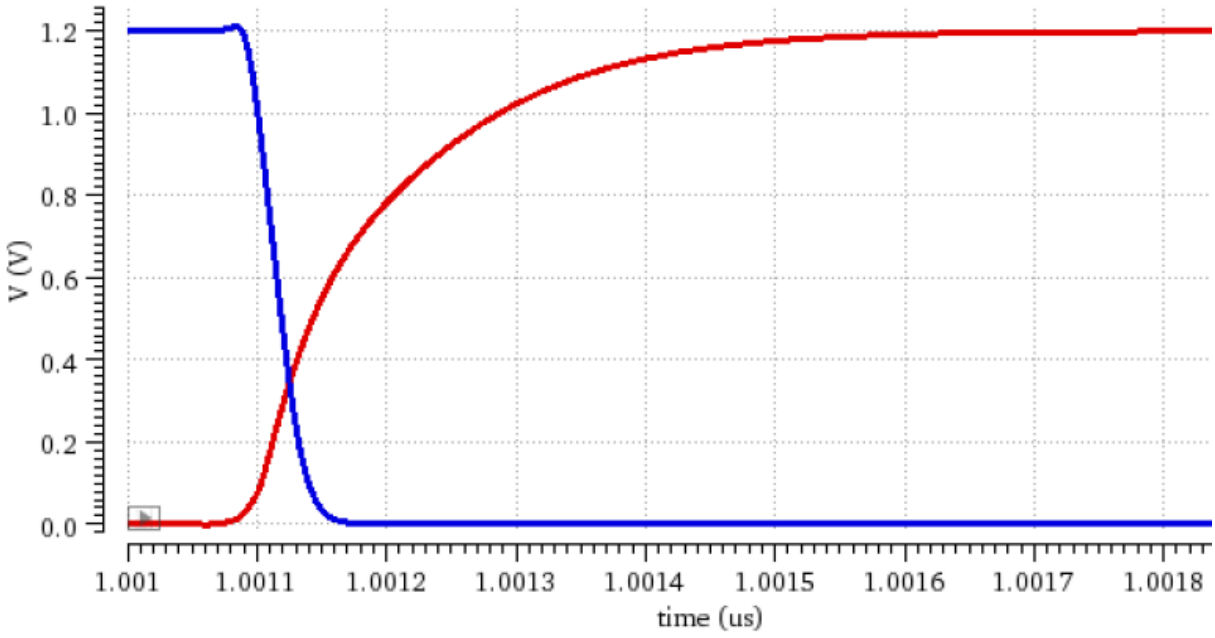


Figure 82 – Lowering the crossing of the switches control signals.

5.3 Biasing Circuit

The biasing circuit used for every current source is shown in figure 83. The right branch functions as the current mirror and the left one generates the biasing voltage V_{CAS} for the cascode transistor that creates the correct V_{DS} for the current source transistor. The reference currents I_{REF} and I_{CAS} for both branches should be created by a precise reference generator circuit independent of temperature and supply voltage and should have the values 4.88 μA and 11.71 μA respectively to achieve the desired biasing. The devices in this cascode configuration match the sizing in table 8 for the current cell cascode configuration.

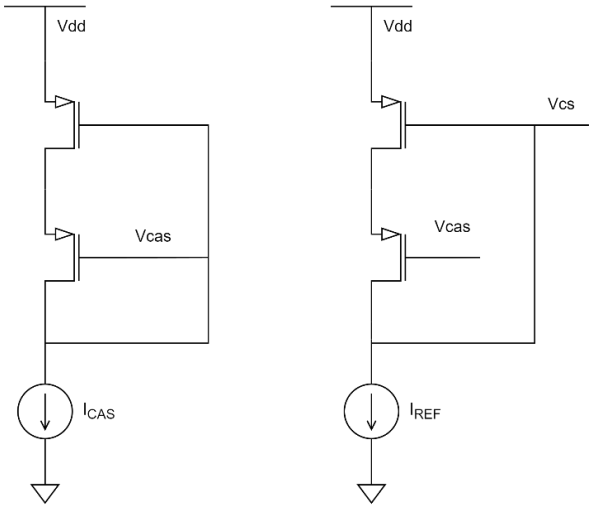


Figure 83 – Biasing circuit.

5.4 Complete DAC Simulation

The full DAC is depicted in figure 84. It is a 10-bit Current Steering DAC with a segmentation of 60%, meaning that the 4 least significant bits are used for binary codification and the other 6 bits have to be converted into thermometer code to be used in unary codification. So, there are the binary groups of 1, 2, 4 and 8 times the LSB current source and 63 unary MSB current sources built with 16 times the LSB current source. In this work the thermometer decoder is a Verilog-A block, and the delay equalizer and the clock are ideal blocks used for simulation purposes only. Simulations of the full DAC with transistor mismatch are very computationally heavy and so only a few were done for illustration. The high-level simulations shown before were done to resolve this issue.

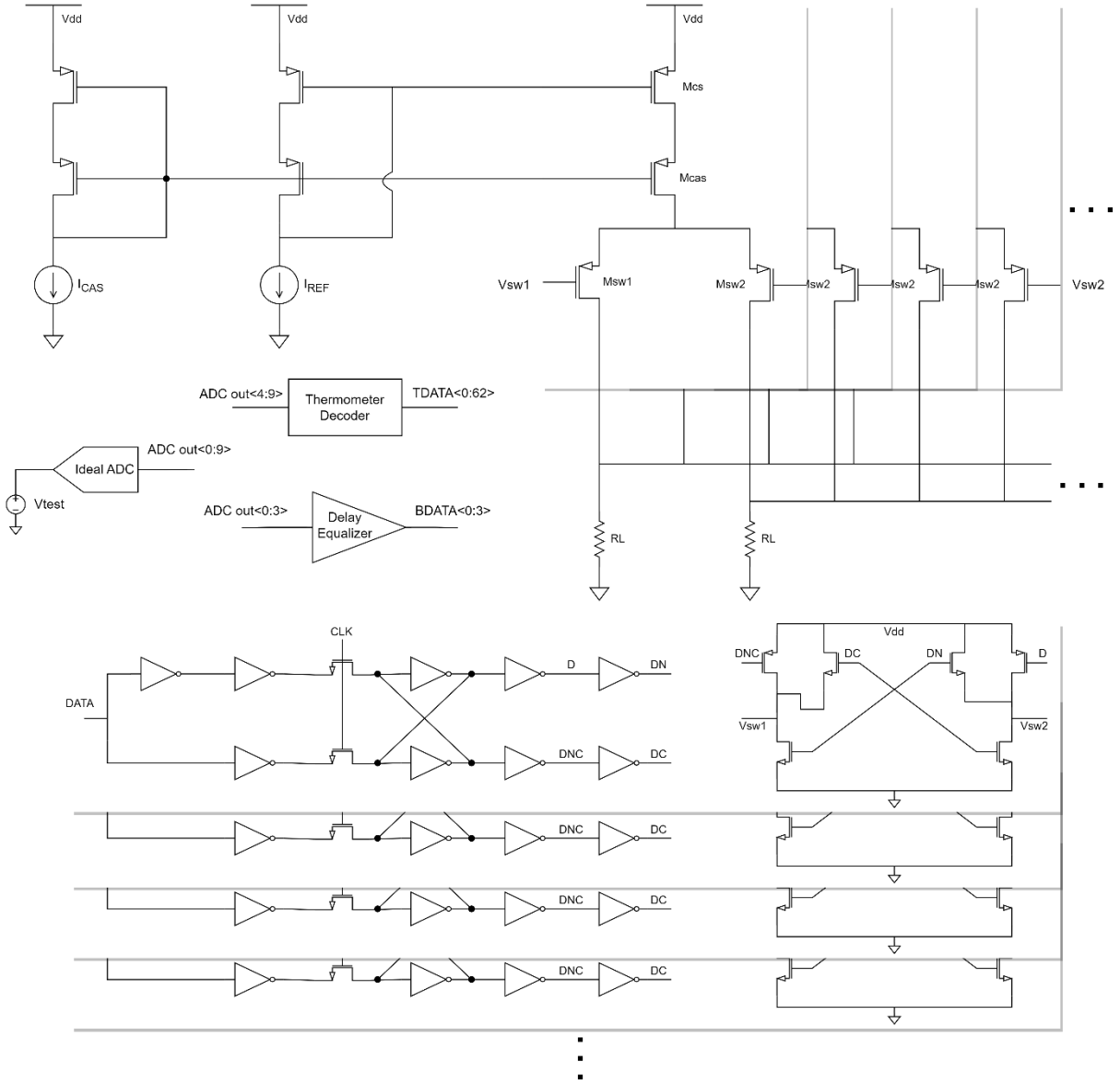


Figure 84 - Full DAC and simulation setup.

Examples of INL and DNL for the typical corner are shown in figures 85 and 86, respectively. These were extracted after a simulation with mismatch between transistors. Figures 87 and 88 show examples of spectrums of the DAC output in the typical conditions, with and without transistor mismatch. Corner simulation results are shown in table 12.

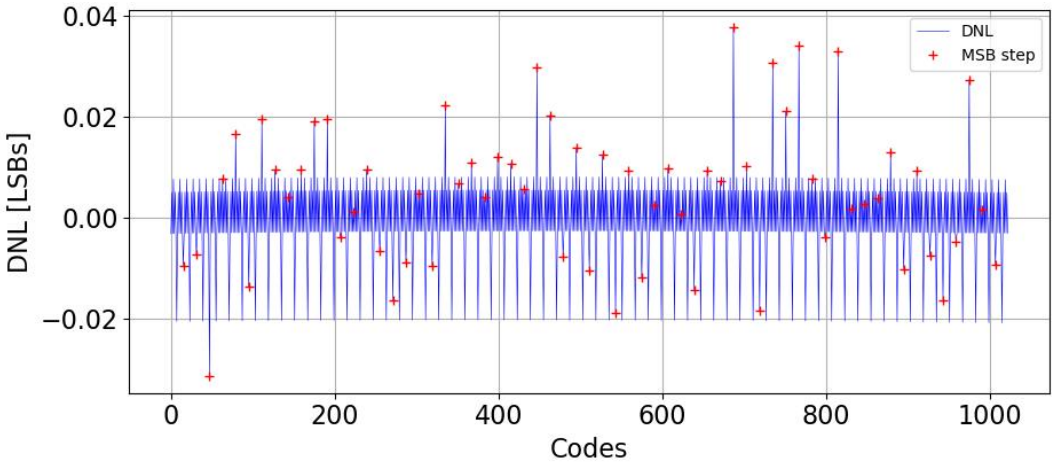


Figure 85 – Example of DNL for typical corner.

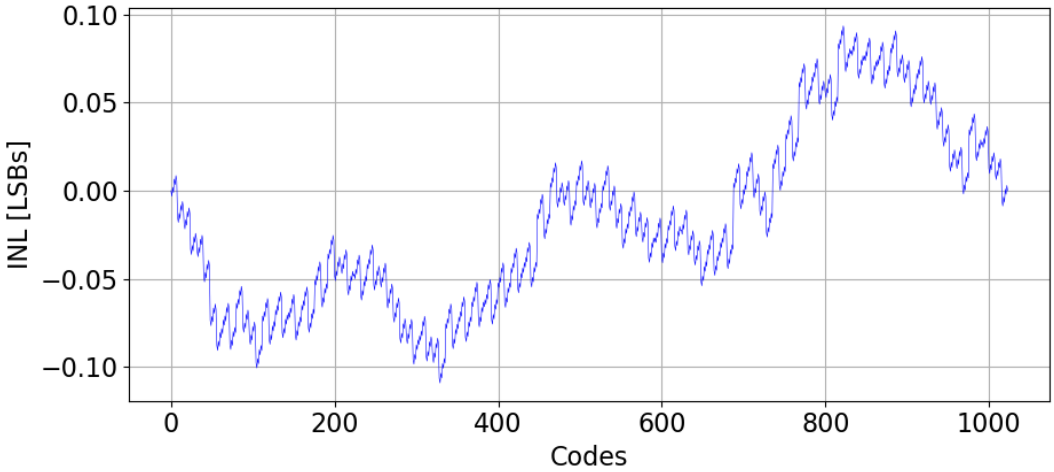


Figure 86 – Example of INL for typical corner.

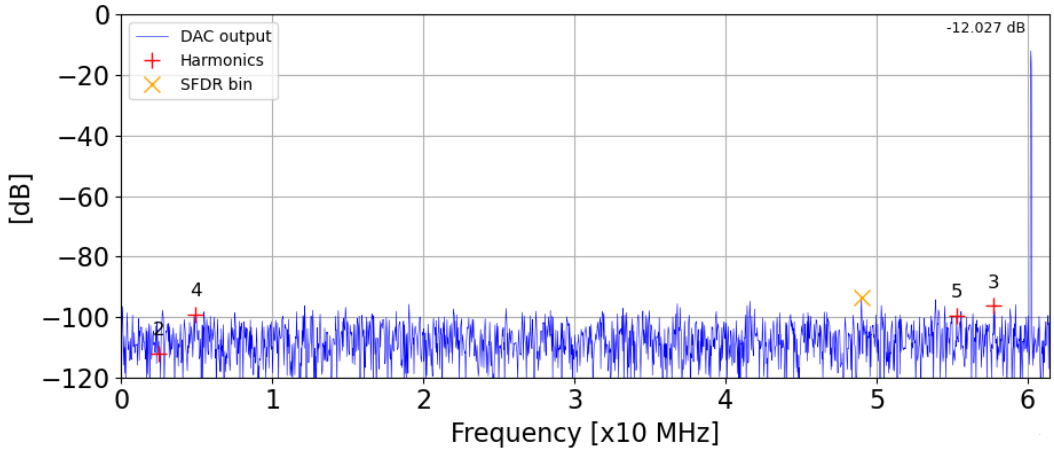


Figure 87 – Spectrum for typical corner C0 without transistor mismatch.

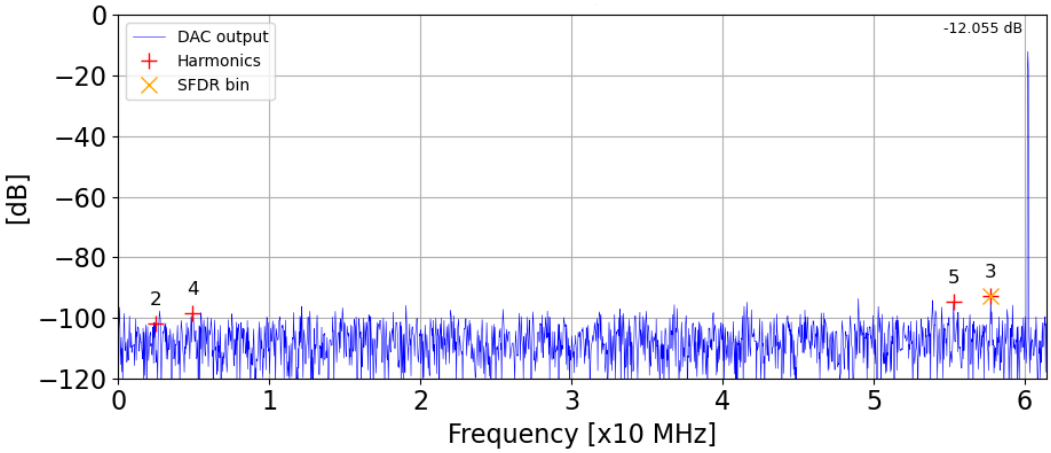


Figure 88 – Example of spectrum for typical corner C0 with transistor mismatch.

Table 12 – Corner simulation results for the complete DAC.

Corner # (Setup in A1 of the appendix)	**SNDR [dB]**	**ENOB [bits]**	**SFDR [dBc]**	**THD [dBc]**	**HD3 [dB]**	**|Swing| [mV]**
0	61.96	10.0	81.73	-80.63	-96.35	250.16
1	61.95	10.0	81.73	-79.76	-94.31	250.00
2	61.91	9.99	79.72	-77.79	-91.08	250.90
3	61.96	10.0	81.73	-80.55	-96.14	250.37
4	61.95	10.0	81.73	-80.29	-95.58	251.18
5	61.89	9.99	78.56	-77.08	-90.01	249.07
6	61.80	9.97	75.93	-75.04	-87.50	249.61
7	61.33	9.90	75.86	-72.79	-88.59	245.24
8	61.96	10.0	81.73	-81.05	-97.67	250.64
9	61.94	10.0	81.73	-79.55	-93.86	249.89
10	61.91	9.99	80.17	-78.09	-91.51	250.62
11	61.96	10.0	81.73	-80.49	-95.95	250.37
12	61.96	10.0	81.73	-80.76	-96.83	251.01
13	61.92	9.99	80.42	-78.32	-91.76	249.47
14	61.87	9.99	78.02	-76.64	-89.47	249.98
15	61.95	10.0	81.73	-80.29	-95.41	250.25
16	61.96	10.0	81.73	-81.02	-97.59	250.78
0 + mmt.	61.89	9.99	81.50	-77.61	-92.83	249.34

Table 12 shows that for every corner the DAC's performance is way above the required metrics. Every corner exhibits an effective number of bits close to 10 bits, SNDR bigger than 61.33 dB, SFDR bigger than 75 dBc, THD of first 4 harmonics lower than -72 dBc, HD3 lower than 87 dB and an absolute value of output swing bigger than 245 mV. There is just one problematic corner which is number 7 where the output swing is much lower than the rest. The high-level simulations show that transistor mismatch would not degrade this spectral performance that much and predicts an SFDR not lower than 72 dBc and an ENOB bigger than 9.94 bits.

6

CONCLUSIONS AND FUTURE WORK

This project was made out of two parts: the development of models for SDDAC's to drive a VCXO, and the design and simulation in a CMOS technology of a 10-bit current steering DAC that could be used in the modelled system.

Regarding the first part, a robust a flexible model was developed that helped take conclusions on which type of SDDAC to use, but further investigation on power consumption, die area cost and even feasibility, especially of the gm-C filter, may be done. The model did not consider any distortion that the gm-C filter may introduce in the system and this aspect should not be disregarded. The models showed clearly that simple digital sigma-delta modulators should be enough to reduce quantization noise to an acceptable level and thus more complex configurations than those in the model may not be considered. The results obtained via high-level simulation may now be used in practice in the real system, starting with transistor level simulation that will further approve or disprove the validity of the model.

The second part of this work showed the design of a current steering DAC with effective number of bits close to 10, an SFDR bigger than 75 dBc at Nyquist and a full-scale voltage close to 0.5 V. The design lacked the layout and post-layout simulations of the circuit which is something to be done in the future. Also, some important blocks like the thermometer decoder, the delay equalizer and clock drivers were considered as ideal in the schematic and simulation and thus they may also be designed in the future.

REFERENCES

[1] Park, J., John Park, A.S.D. and Mackay, S., 2003. Practical data acquisition for instrumentation and control systems. Newnes, pp. 124.

[2] Crystek, "VCXO Ultra-Low Phase Noise Oscillators", CVHD-950 datasheet, 26 Aug. 2019.

[3] Schreier, R. and Temes, G.C., 2005. Understanding delta-sigma data converters (Vol. 74). Piscataway, NJ: IEEE press.

[4] Santina, M.S. and Stubberud, A.R., 2005. Basics of sampling and quantization. In Handbook of networked and embedded control systems, pp. 45-69. Birkhäuser Boston.

[5] José, M. and Del Rio, R., 2013. CMOS sigma-delta converters: Practical design guide.

[6] W. L. Lee, "A novel higher-order interpolative modulator topology for high resolution oversampling A/D converters," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1987.

[7] Kiss, P., Arias, J. and Li, D., 2003, May. Stable high-order delta-sigma DACs. In Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS'03. (Vol. 1, pp. I-I). IEEE.

[8] Ameur, N.B. and Loulou, M., 2008, March. Design of efficient digital interpolation filters and sigma-delta modulator for audio DAC. In 2008 3rd International Conference on Design and Technology of Integrated Systems in Nanoscale Era (pp. 1-7). IEEE.

[9] Xu, X., Temes, G. and Schreier, R., 1992, May. The implementation of dual-truncation sigma delta D/A converters. In [Proceedings] 1992 IEEE International Symposium on Circuits and Systems (Vol. 2, pp. 597-600). IEEE.

[10] Venturini, Giuseppe. (2016) python-deltasigma v0.2.2 documentation. Available at: https://python-deltasigma.readthedocs.io/en/latest/ (Accessed: Dec. 2021).

[11] Texas Instruments, “Ultra Low-Noise JESD204B Compliant Clock Jitter Cleaner With Dual Loop PLLs”, LMK0482x datasheet, revised May 2020.

[12] C.-H. Lin, K. Bult, "A 10-b, 500MSample/s CMOS DAC in 0.6mm^2," IEEE JSSC, Vol 33, pp. 1948-1958, Dec 1998.

[13] S. Luschas and H. -. Lee, "Output impedance requirements for DACs," Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03., 2003, pp. I-I, doi: 10.1109/ISCAS.2003.1205700.

[14] A. van den Bosch, M. Steyaert and W. Sansen, "SFDR-bandwidth limitations for high speed high resolution current steering CMOS D/A converters," ICECS'99. Proceedings of ICECS '99. 6th IEEE International Conference on Electronics, Circuits and Systems (Cat. No.99EX357), 1999, pp. 1193-1196 vol.3, doi: 10.1109/ICECS.1999.814383.

[15] Fang-Jie Luo, Yong-Sheng Yin, Shang-Quan Liang and Ming-Lun Gao, "Current switch driver and current source designs for high-speed current-steering DAC," 2008 2nd International Conference on Anti-counterfeiting, Security and Identification, 2008, pp. 364-367, doi: 10.1109/IWASID.2008.4688421.

[16] A. van den Bosch, M. A. F. Borremans, M. S. J. Steyaert and W. Sansen, "A 10-bit 1-GSample/s Nyquist current-steering CMOS D/A converter," in IEEE Journal of Solid-State Circuits, vol. 36, no. 3, pp. 315-324, March 2001, doi: 10.1109/4.910469.

[17] A. Van den Bosch, M. Steyaert and W. Sansen, "An accurate statistical yield model for CMOS current-steering D/A converters," 2000 IEEE International Symposium on Circuits and Systems (ISCAS), 2000, pp. 105-108 vol.4, doi: 10.1109/ISCAS.2000.858699.

[18] J. Deveugele and M. S. J. Steyaert, "A 10-bit 250-MS/s binary-weighted current-steering DAC," in IEEE Journal of Solid-State Circuits, vol. 41, no. 2, pp. 320-329, Feb. 2006, doi: 10.1109/JSSC.2005.862342.

[19] Lakshmikumar, Kadaba R., Robert A. Hadaway and Miles A. Copeland. “Characterisation and modeling of mismatch in MOS transistors for precision analog design.” IEEE Journal of Solid-state Circuits 21 (1986): 1057-1066.

[20] M. J. M. Pelgrom, A. C. J. Duinmaijer and A. P. G. Welbers, "Matching properties of MOS transistors," in IEEE Journal of Solid-State Circuits, vol. 24, no. 5, pp. 1433-1439, Oct. 1989, doi: 10.1109/JSSC.1989.572629.

[21] Song, B.-S. (2012). Micro CMOS Design (1st ed.). CRC Press. https://doi.org/10.1201/b11192

[22] Razavi, Behzad. 2001. Design of analog CMOS integrated circuits. Boston, MA: McGraw-Hill.

[23] A. Demir, A. Mehrotra and J. Roychowdhury, "Phase noise in oscillators: a unifying theory and numerical methods for characterization," in IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, vol. 47, no. 5, pp. 655-674, May 2000, doi: 10.1109/81.847872.

[24] Poore, Rick. "Phase Noise and Jitter Rick Poore Agilent EEsof EDA." (2001).

[25] T.-Y. Lo and C.-C. Hung, 1V CMOS Gm-C Filters: Design and Applications, Analog Circuits and Signal Processing, c Springer Science+Business Media B.V. 200.

[26] S. -Y. Lee and C. -J. Cheng, "Systematic Design and Modeling of a OTA-C Filter for Portable ECG Detection," in IEEE Transactions on Biomedical Circuits and Systems, vol. 3, no. 1, pp. 53-64, Feb. 2009, doi: 10.1109/TBCAS.2008.2007423.

[27] N. H. Hamid, A. F. Murray and S. Roy, "Time-Domain Modeling of Low-Frequency Noise in Deep-Submicrometer MOSFET," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 55, no. 1, pp. 245-257, Feb. 2008, doi: 10.1109/TCSI.2007.910543.

[28] Fang-Jie Luo, Yong-Sheng Yin, Shang-Quan Liang and Ming-Lun Gao, "Current switch driver and current source designs for high-speed current-steering DAC," 2008 2nd International Conference on Anti-counterfeiting, Security and Identification, 2008, pp. 364-367, doi: 10.1109/IWASID.2008.4688421.

APPENDIX

A1. Corners Simulation Setup

Table 13 – Corners simulation setup.

Corner #	Temperature [°C]	Vdd [V]	NMOS	PMOS	Resistor
0	27	1.2	Typical	Typical	Typical
1	-40	1.08	Fast	Fast	Max
2	125	1.08	Fast	Fast	Max
3	-40	1.32	Fast	Fast	Max
4	125	1.32	Fast	Fast	Max
5	-40	1.08	Slow	Slow	Min
6	125	1.08	Slow	Slow	Min
7	-40	1.32	Slow	Slow	Min
8	125	1.32	Slow	Slow	Min
9	-40	1.08	Slow	Fast	Max
10	125	1.08	Slow	Fast	Max
11	-40	1.32	Slow	Fast	Max
12	125	1.32	Slow	Fast	Max
13	-40	1.08	Fast	Slow	Min
14	125	1.08	Fast	Slow	Min
15	-40	1.32	Fast	Slow	Min
16	125	1.32	Fast	Slow	Min

A2. Sigma-Delta Modulators Spectrums

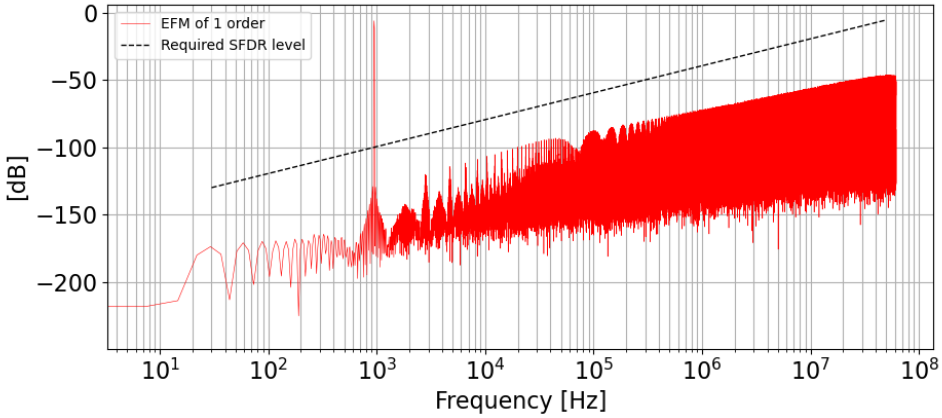


Figure 89 – 48b to 1b EFM1 full-scale.

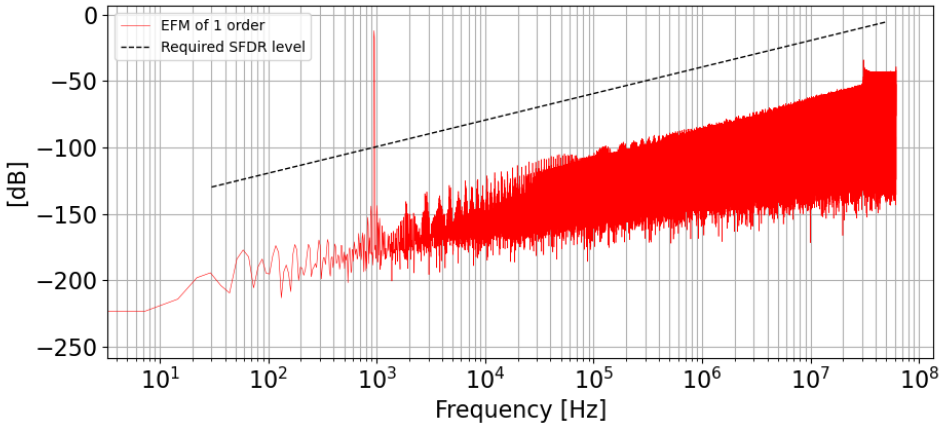


Figure 90 - 48b to 1b EFM1 half-scale.

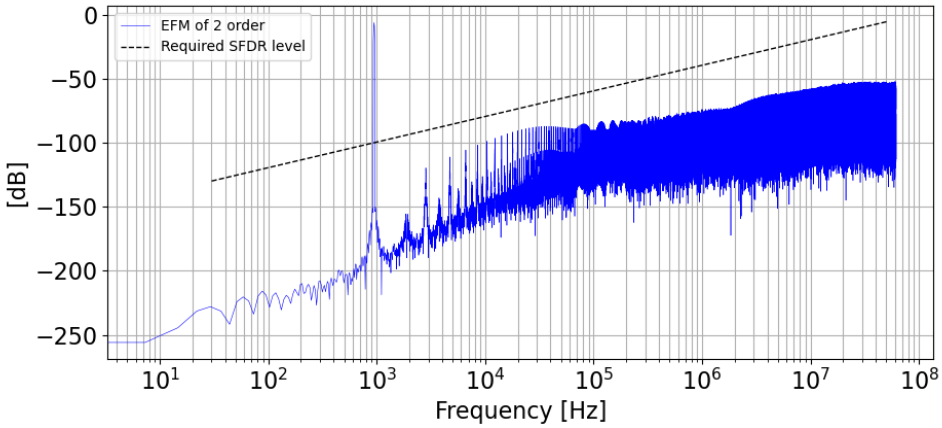


Figure 91 – 48b to 1b EFM2 full-scale.

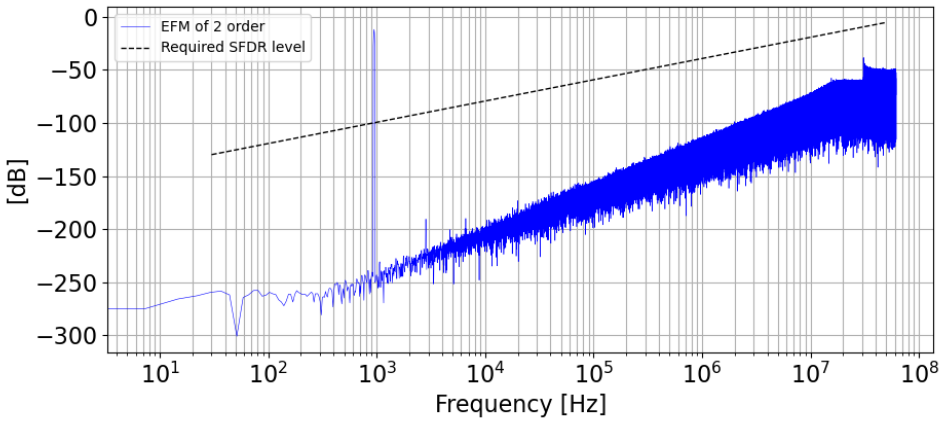


Figure 92 – 48b to 1b EFM2 half-scale.

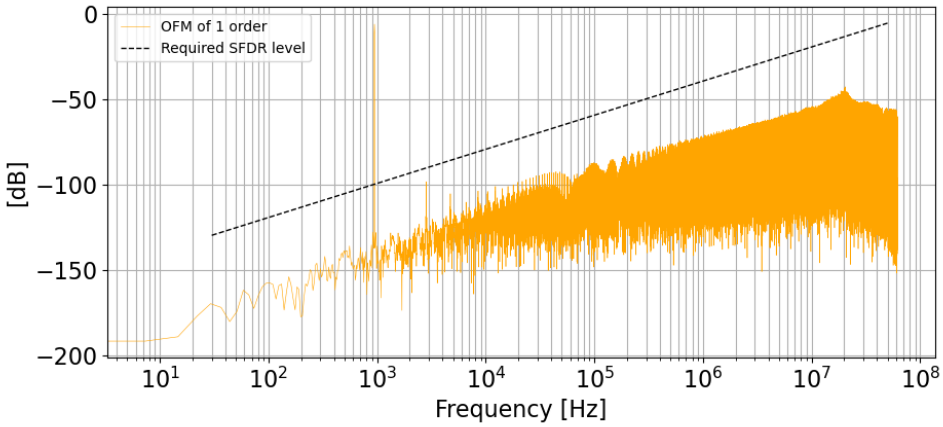


Figure 93 – 48b to 1b 48b to 1b OFM1 full-scale.

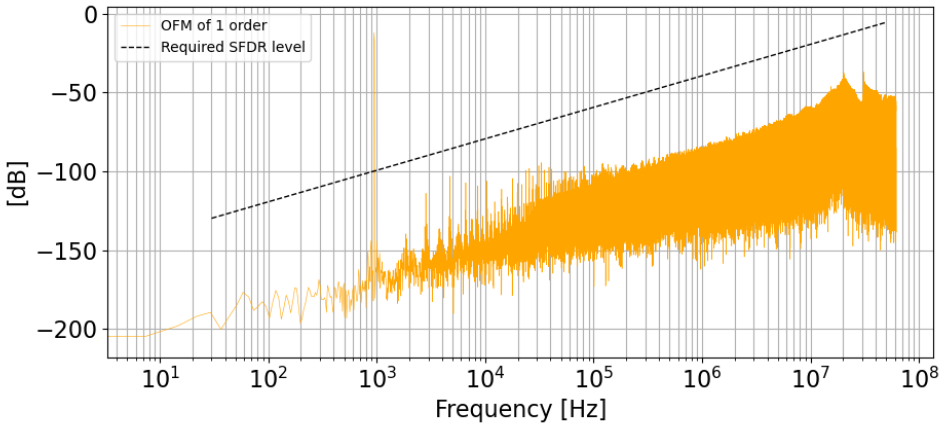


Figure 94 – 48b to 1b OFM1 half-scale.

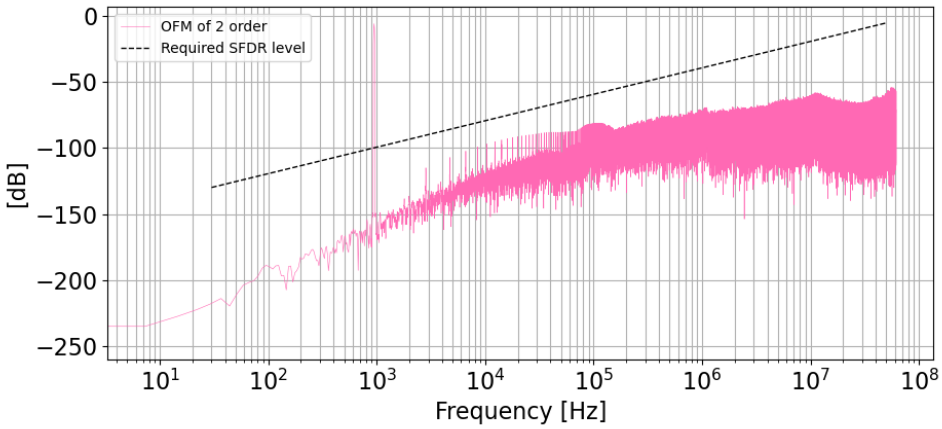


Figure 95 – 48b to 1b OFM2 full-scale.

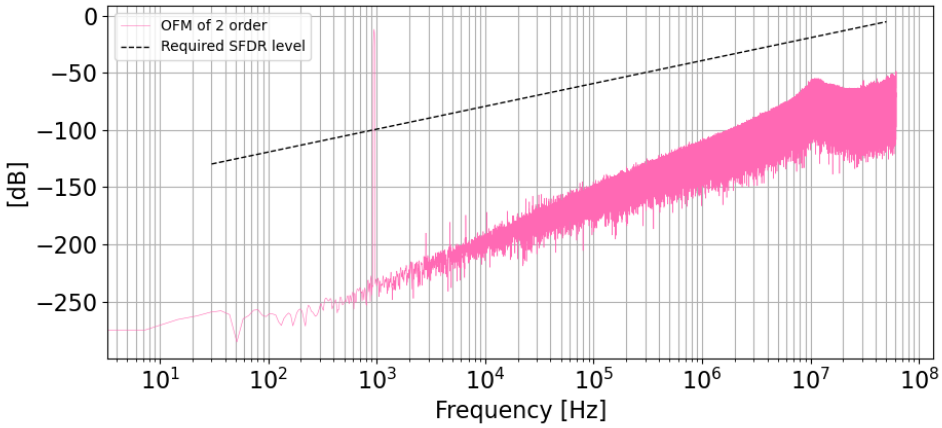


Figure 96 – 48b to 1b OFM2 half-scale.

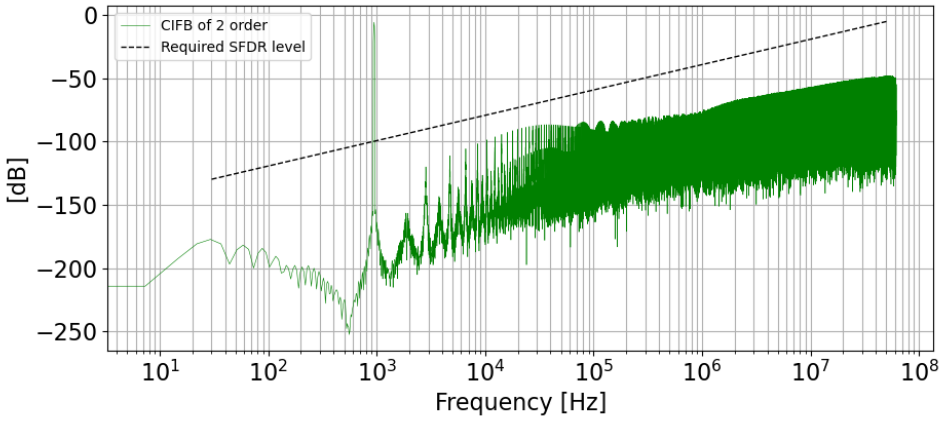


Figure 97 – 48b to 1b CIFB full-scale.

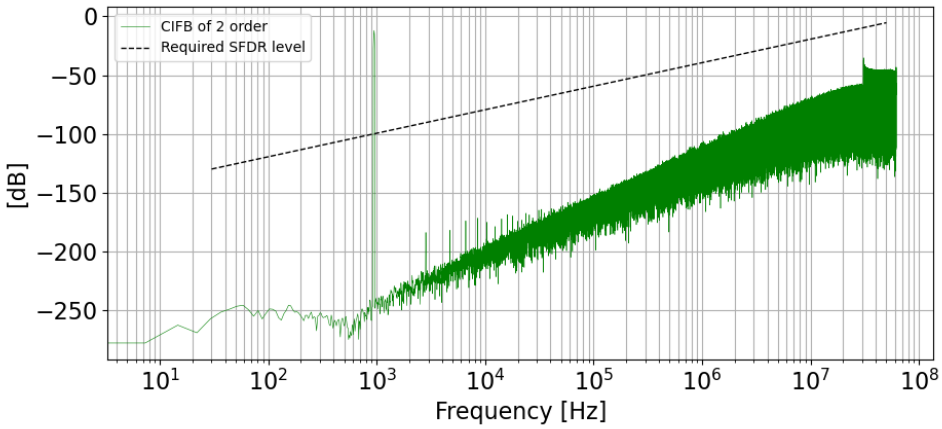


Figure 98 – 48b to 1b CIFB half-scale.

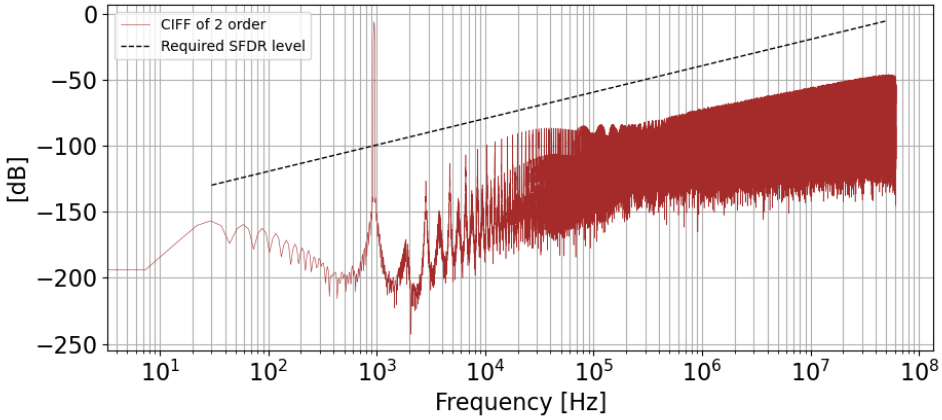


Figure 99 – 48b to 1b CIFF full-scale.

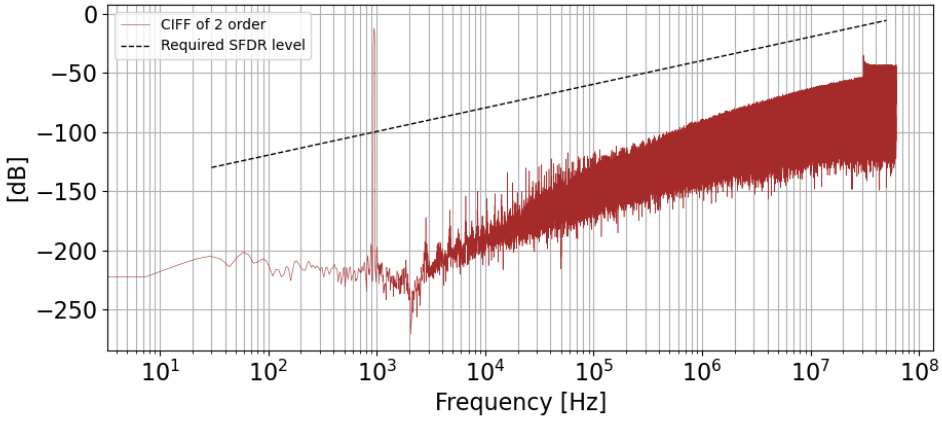


Figure 100 – 48b to 1b CIFF half-scale.

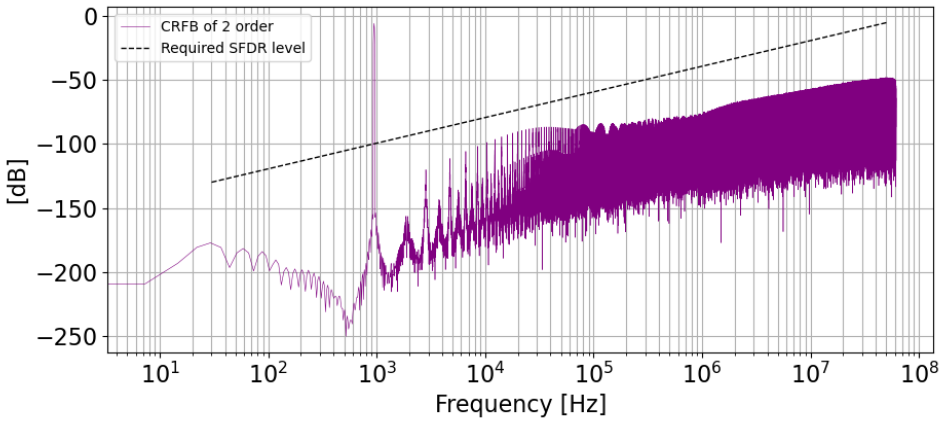


Figure 101 – 48b to 1b CRFB full-scale.

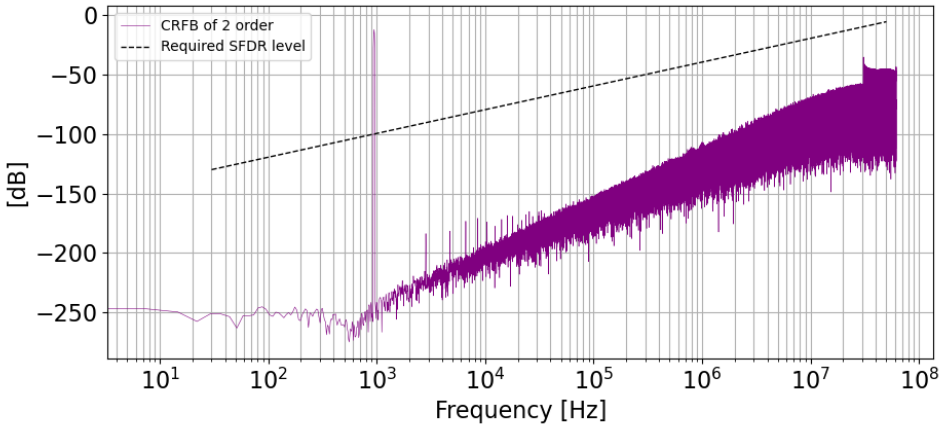


Figure 102 – 48b to 1b CRFB half-scale.

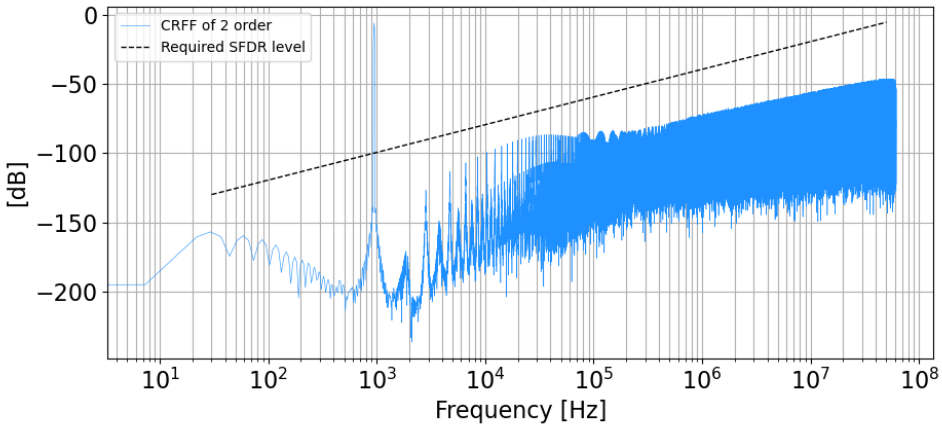


Figure 103 – 48b to 1b CRFF full-scale.

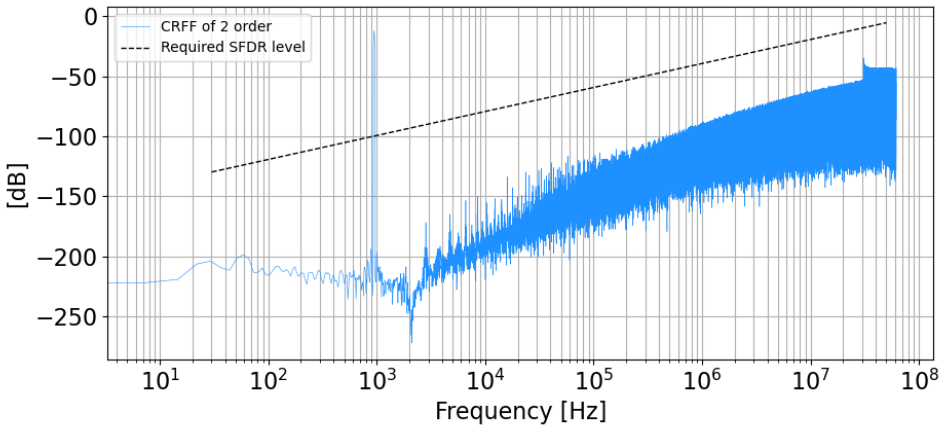


Figure 104 – 48b to 1b CRFF half-scale.

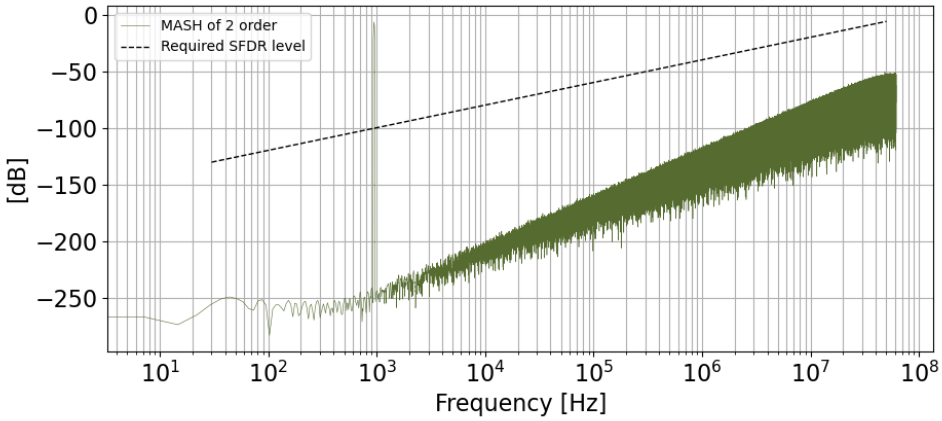


Figure 105 – 48b to 2b MASH2 full-scale.

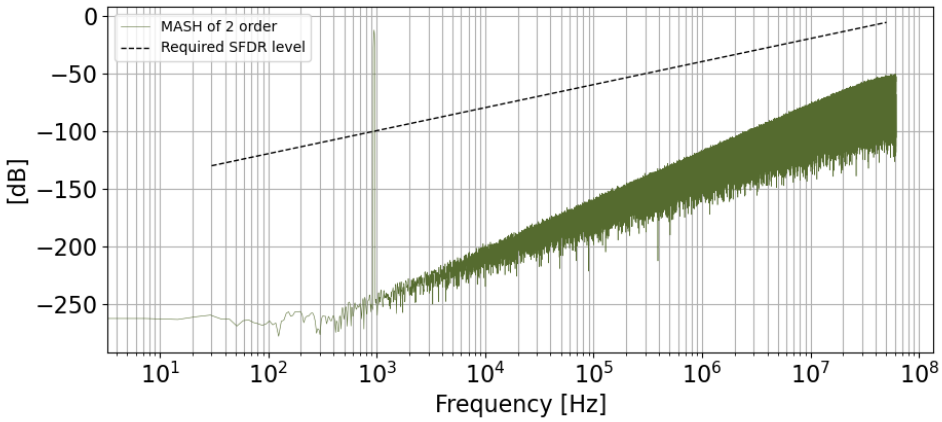


Figure 106 – 48b to 2b MASH2 half-scale.

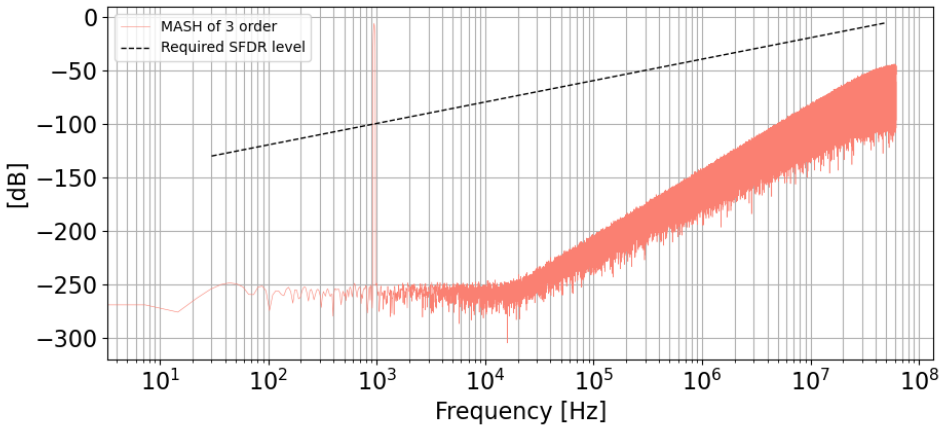


Figure 107 – 48b to 3b MASH3 full-scale.

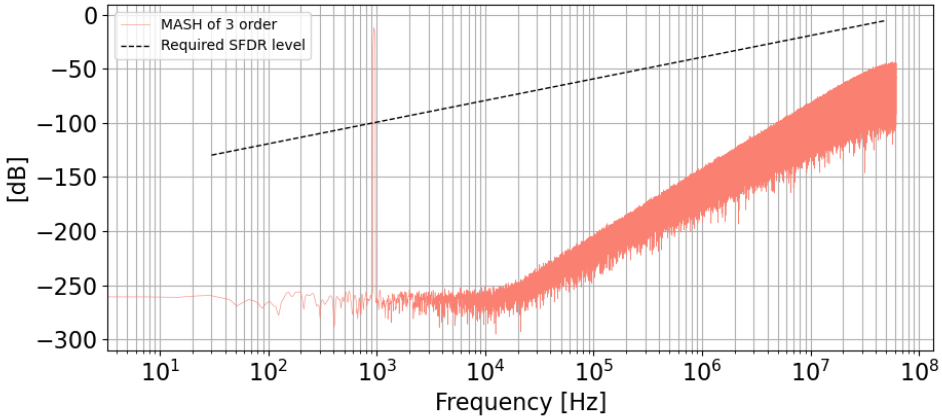


Figure 108 – 48b to 3b MASH3 half-scale.

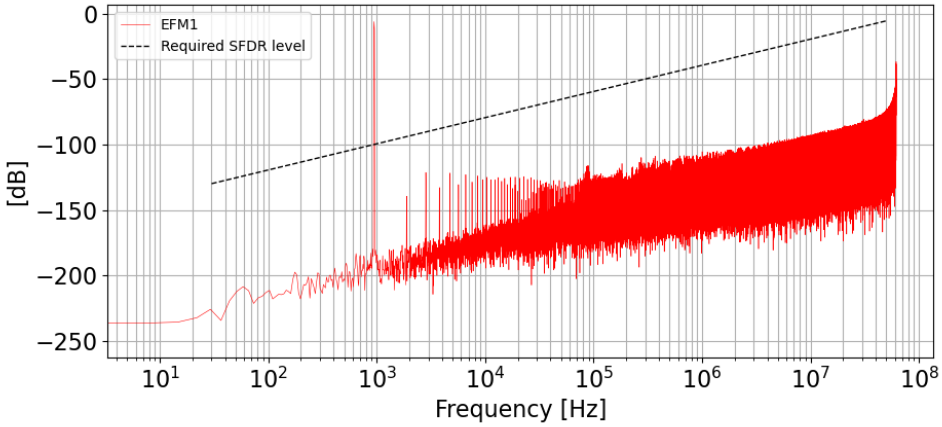


Figure 109 – 48b to 5b EFM1 full-scale.

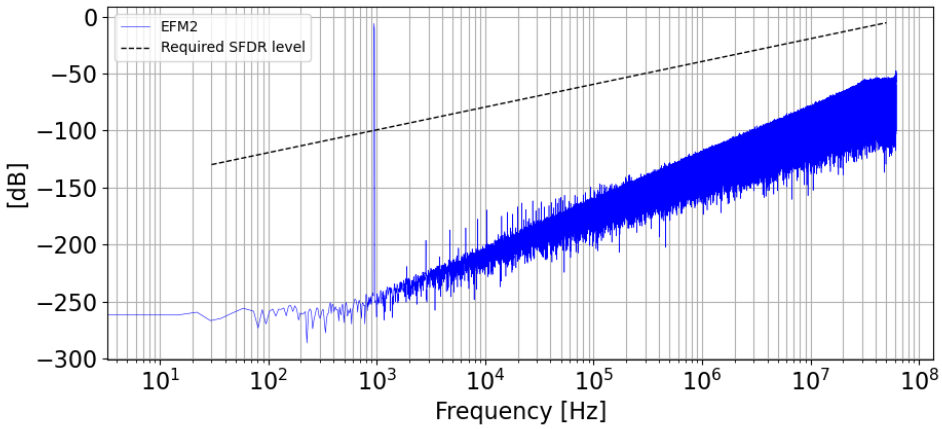


Figure 110 – 48b to 5b EFM2 full-scale.

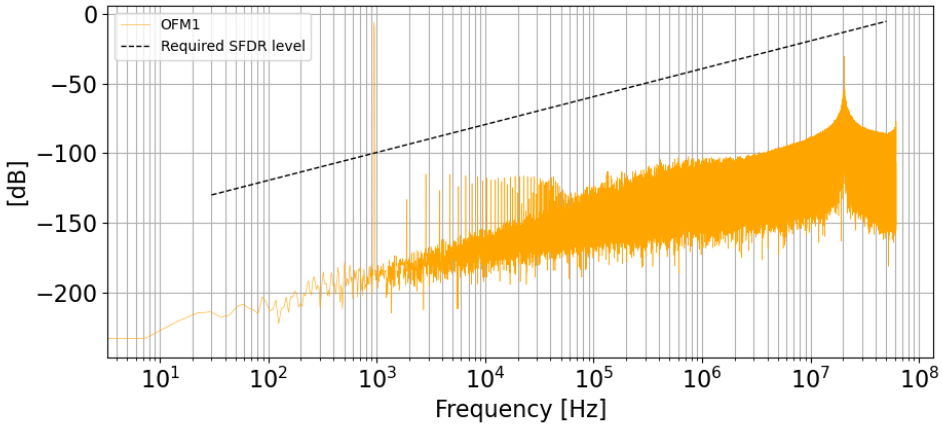


Figure 111 – 48b to 5b OFM1 full-scale.

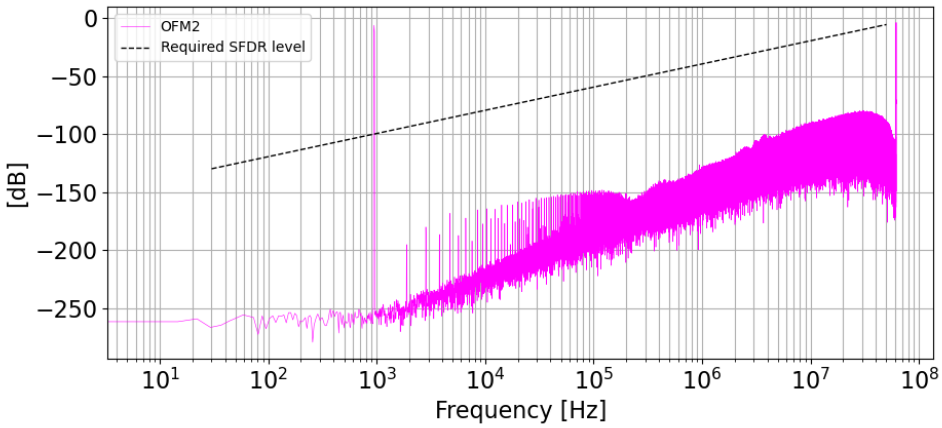


Figure 112 – 48b to 5b OFM2 full-scale.

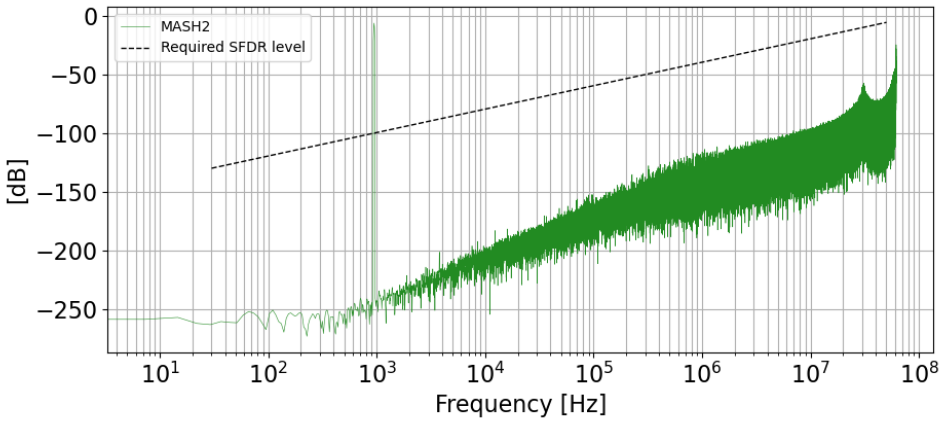


Figure 113 – 48b to 6b MASH2 full-scale.

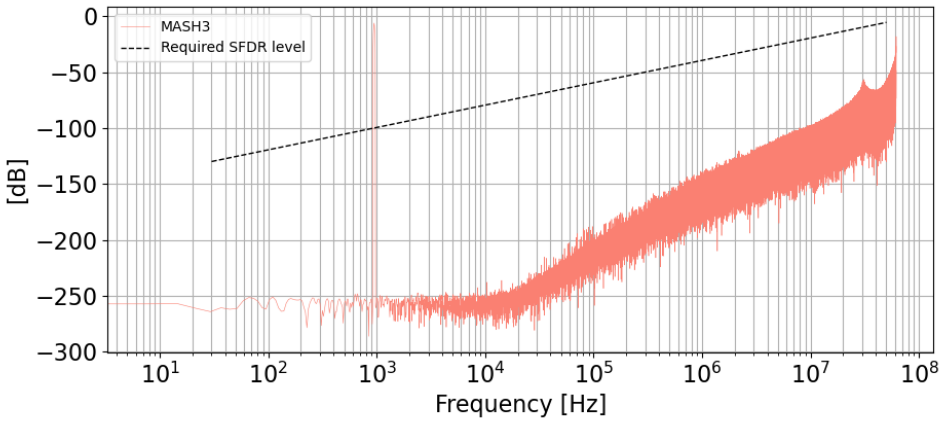


Figure 114 – 48b to 7b MASH3 full-scale.

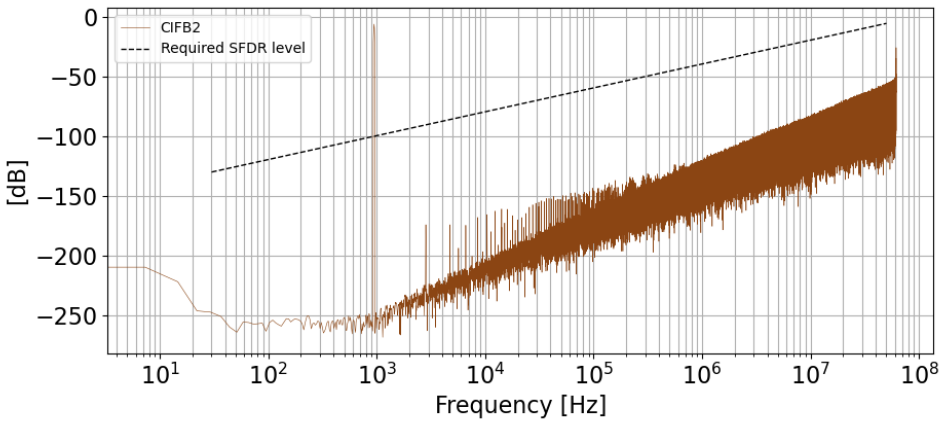


Figure 115 – 48b to 5b CIFB2 full-scale.

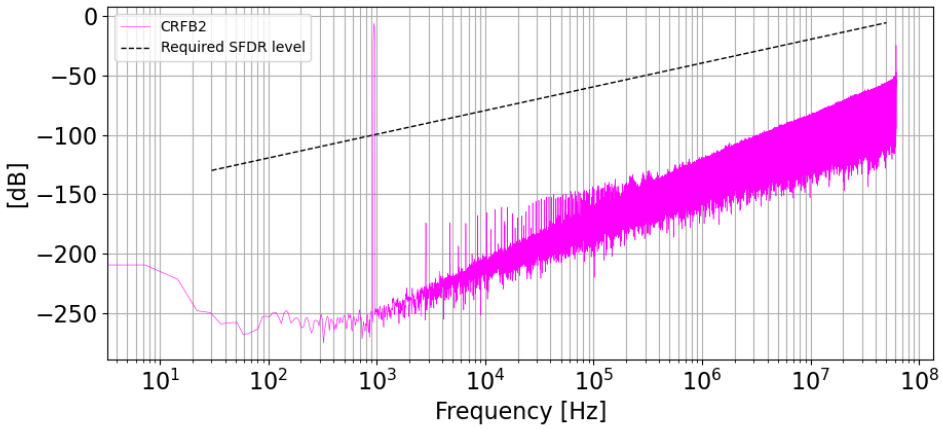


Figure 116 – 48b to 5b CRFB2 full-scale.

A3. Sigma-Delta Modulator Python Code

```
import deltasigma as ds
import numpy as np
import aux_sdm as aux
import warnings

class DSDM:
    def __init__(self, top, osr, order, qbits, in_bits, qtype = 1,
                 H_inf = 1.5, max_err = 0.01, gain = 1):
        #oversampling ratio
        self.osr = osr
        #modulator's order
        self.order = order
        #maximum alowed error when converting decimals to sums of powers of 2
        self.max_err = max_err
        #quantization bits
        self.qbits = qbits
        #infinity gain of NTF
        self.H_inf = H_inf
        #input word length
        self.in_bits = in_bits
        #modulators topology
        self.top = top
        #quantizer type: 1 - mid-rise, 2 - mid-thread
        self.qtype = qtype
        #input attenuation to prevent clipping
        self.gain = gain
        print("\n___Digital Sigma Delta Modulator___\n" )
        print("Topology:", top)
        print("Order =", order)
        print("Single stage quantization =", qbits, "bits")
        if qtype == 1:
            print("Quantization type: mid-rise (1 bit -> -1 or 1)")
        elif qtype == 2:
            print("Quantization type: mid-thread (1 bit -> -1, 0 or 1 -> 1.5
bits)")
        else:
            raise Exception("Quantizer types allowed are 1 or 2.")
        print("Input word:", in_bits, "bits")
        print("Signed words with 1 extra bit as sign bit\n")
```

```
    """
    mod()
    Receives an input digital sequence un, passes it through a gain block,
    creates a digital modulator with the chosen topology
    and output the truncated, modulated digital signal
    """
    def mod(self, un):
        if self.gain != 1:
            gain_p2 = self.sum_of_powers_of_two(self.gain)
            un = (un * sum(gain_p2)).astype(np.int64)
        if self.top == "MASH":
            return self.MASH(un)
        elif self.top == "CIFB":
            return self.CIFB(un)
        elif self.top == "CRFB":
            return self.CRFB(un)
        elif self.top == "CIFF":
            return self.CIFF(un)
        elif self.top == "CRFF":
            return self.CRFF(un)
        elif self.top == "OFM":
            if self.order == 2:
                return self.OFM2(un)
            elif self.order == 1:
                return self.OFM1(un)
            else:
                raise Exception("order > 2 not allowed for this topology.")
                return np.zeros(un.size)
        elif self.top == "EFM":
            if self.order == 2:
                return self.EFM2(un)
            elif self.order == 1:
                return self.EFM(un)
            else:
                raise Exception("order > 2 not allowed for this topology.")
                return np.zeros(un.size)
        elif self.top == "HKEFM":
            return self.HKEFM(un)
        else:
            raise Exception("Topology not recognized.")
            return np.zeros(un.size)
    """
    quantize()
    Receives a number v, quantizes it by right shifting by s,
    formats it to the right quantizer output, and applies a limiter
    """
    def quantize(self, v, s):
        #truncation
        y = v >> s
        #convert ...-3 to -5, -2 to -3, 0 to 1; 2 to 3; 3 to 5... (odd ints.)
        if self.qtype == 1:
            if y == 0:
                y = 1
            else:
                y = int(y + y/abs(y) * (abs(y)-1))
        #limiter
        MAX = abs((2**(self.qbits-1))-1)
        if self.qbits == 1:
            MAX = 1
        if y > MAX:
            y = MAX
        if y < -MAX:
            y = -MAX
        return y
```

```
    """
    nearest_power_of_two()
    Receives a number x and outputs the closest power of 2
    """
    def nearest_power_of_two(self, x):
        import numpy as np
        if x == 0:
            return 0
        if x < 0:
            sign = -1
        else:
            sign = 1
        x = abs(x)
        up = 2**np.ceil(np.log2(x))
        down = 2**np.floor(np.log2(x))
        error_up = abs(up - x)
        error_down = abs(down - x)
        if error_up < error_down:
            return sign * up
        else:
            return sign * down

    """
    sum_of_powers_of_two()
    Receives a number x and outputs an array with powers of 2
    which added approximate to x with maximum error max_error
    """
    def sum_of_powers_of_two(self, x):
        import numpy as np
        result = [self.nearest_power_of_two(x)]
        if sum(result) == 0:
            if x == 0:
                error = 0
            else:
                error = (x - sum(result))/x
        else:
            error = (x - sum(result))/sum(result)
        while(abs(error) > self.max_err):
            diff = x - sum(result)
            result.append(self.nearest_power_of_two(diff))
            if sum(result) == 0:
                if x == 0:
                    error = 0
                else:
                    error = (x - sum(result))/x
            else:
                error = (x - sum(result))/sum(result)
        return np.array(result)

    """
    shifts()
    Receives an array with powers of 2
    and outputs [+ or - for << or >>, shift value]
    """
    def shifts(self, nums):
        import numpy as np
        return [[int(n/abs(n)), int(np.log2(abs(n)))]
                if n != 0 else [0, 0] for n in nums]

    """
    coef2shifts()
    Receives arrays of coefficients a, b, c, d
    Outputs arrays with shifts and
    arrays with the rounded coefficients
    """
```

```
    def coef2shifts(self, a, b, c, g = [0]):
        A = []
        B = []
        C = []
        G = []

        for i, aa in enumerate(a):
            Aa = self.sum_of_powers_of_two(aa)
            A.append(self.shifts(Aa))
            a[i] = sum(Aa)
        for i, bb in enumerate(b):
            Bb = self.sum_of_powers_of_two(bb)
            B.append(self.shifts(Bb))
            b[i] = sum(Bb)
        for i, cc in enumerate(c):
            Cc = self.sum_of_powers_of_two(cc)
            C.append(self.shifts(Cc))
            c[i] = sum(Cc)
        for i, gg in enumerate(g):
            Gg = self.sum_of_powers_of_two(gg)
            G.append(self.shifts(Gg))
            g[i] = sum(Gg)

        print("Coefficients:")
        print("a's =", a)
        print("b's =", b)
        print("c's =", c)
        print("g's =", g)

        return A, B, C, G, a, b, c, g

    """
    get_a()
    Get the number a which is the closest,
    smaller relatively prime number to M = 2**bits
    """
    def get_a(self):
        #number of levels at input
        M = 2**self.in_bits
        #loop to finf a such that M - a is prime
        for a in range(10000):
            if aux.is_prime(M-a):
                return a
        return 1

    """
    OFM1()
    Models a basic first order output feedback modulator
    """
    def OFM1(self, un):
        #number of samples
        ns = un.size
        #internal initial conditions
        i1 = 1
        i2 = 1
        #output array
        yn = np.zeros(ns).astype(np.int64)
        #auxiliary arrays for checking number of intermidiate bits
        i1n = np.zeros(ns).astype(np.int64)
        i2n = np.zeros(ns).astype(np.int64)
        #quantizer specs
        b = self.in_bits+1 #expected quantizer input word length
        s = b - self.qbits #right shift
        #simulation over time
        for j in range(1, ns):
```

```
            #quantization
            y = self.quantize(i2, s)
            #compute internal states
            i2 += i1
            i1 = un[j] - (y << (s-1))
            #outpuy
            yn[j-1] = y
            #auxiliary arrays for checking number of intermidiate bits
            i1n[j] = i1
            i2n[j] = i2

        print("---> First order OFM")
        print("     Number of bits before first integrator:",
len(bin(max(abs(i1n))))-1)
        print("     Number of bits before quantization:",
len(bin(max(abs(i2n))))-1)
        print("     Left shifting by", s-1, "bits to feedback the output")
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("     Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        return yn

    """
    OFM2()
    Models a basic second order output feedback modulator
    """
    def OFM2(self, un):
        #number of samples
        ns = un.size
        #internal initial conditions
        i1 = 1
        i2 = 1
        i3 = 1
        #auxiliary arrays for checking number of intermidiate bits
        i1n = np.zeros(ns).astype(np.int64)
        i2n = np.zeros(ns).astype(np.int64)
        i3n = np.zeros(ns).astype(np.int64)
        #output array
        yn = np.zeros(ns).astype(np.int64)
        #quantizer specs
        b = self.in_bits+2 #expected quantizer input word length
        s = b - self.qbits #right shift
        #simulation over time
        for j in range(1, ns):
            #quantization
            y = self.quantize(i3, s)
            #compute internal states
            i3 += i2 - (y << s)
            i2 += i1
            i1 = un[j] - (y << (s-2))
            #output
            yn[j-1] = y
            #auxiliary arrays for checking number of intermidiate bits
            i1n[j] = i1
            i2n[j] = i2
            i3n[j] = i3

        print("---> Second order OFM")
        print("     Number of bits before first integrator:",
len(bin(max(abs(i1n))))-1)
```

```
        print("     Number of bits before second integrator:",
len(bin(max(abs(i2n))))-1)
        print("     Number of bits before quantization:",
len(bin(max(abs(i3n))))-1)
        print("     Left shifting by", s-2, "bits to feedback the output")
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("     Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        return yn

    """
    MASH()
    Creates a model for a EMF MASH modulator
    """
    def MASH(self, un = []):
        #number of samples
        ns = un.size
        #cancelation logic output array
        w = np.zeros(ns).astype(np.int64)
        #EMF first stage
        print("-> Stage", 1)
        y1, e = self.HKEFM(un, mash = True)
        print("     Number of bits in error to next stage:",
len(bin(max(abs(e.astype(np.int64)))))-1)
        print("     Number of bits after cancelation logic:",
len(bin(max(abs(y1.astype(np.int64)))))-1)
        #other stages
        for i in range(self.order-1):
            print("-> Stage", 2+i)
            y2, e = self.HKEFM(e, mash = True)
            print("     Number of bits in error to next stage:",
len(bin(max(abs(e.astype(np.int64)))))-1)
            for j in range(i+1):
                #cancelation logic filter
                y2 = self.CL_EFM(y2)
            print("     Number of bits after cancelation logic:",
len(bin(max(abs(y2.astype(np.int64)))))-1)
            #output summations
            w += y2
        yn = y1 + w
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        return yn

    """
    CL_EMF1()
    Implements a filter for the cancelation logic in a EMF1 MASH modulator
    """
    def CL_EFM(self, y):
        #number of samples
        ns = y.size
        #filter output array
        w = np.zeros(ns).astype(np.int64)
        #filtering over time
```

```
        for n in range(1, ns):
            w[n] = y[n] - y[n-1]
        return w

    """
    EMF1()
    Creates a model of a EMF1 modulator
    """
    def EFM(self, un = []):
        #number of samples
        ns = un.size
        #output array
        yn = np.zeros(ns).astype(np.int64)
        #auxiliary arrays for checking number of intermidiate bits
        vn = np.zeros(ns).astype(np.int64)
        #error array
        e = np.zeros(ns).astype(np.int64)
        #quantizer specs
        b = self.in_bits+1 #expected quantizer input word length
        s = b - self.qbits #right shift
        #simulation over time
        for n in range(1, ns):
            u = un[n]
            #accumulator
            v = u + e[n-1]
            #quantization
            y = self.quantize(v, s)
            #compute error feedback
            e[n] = v - (y << s-1)
            #output
            yn[n] = y
            #auxiliary arrays for checking number of intermidiate bits
            vn[n] = v

        print("---> First order EFM")
        print("     Number of bits before truncation:", len(bin(max(abs(vn))))-1)
        print("     Number of bits of quantization error:",
len(bin(max(abs(e))))-1)
        print("     Left shifting by", s, "bits to feedback the output")
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("     Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        return yn

    """
    HKEMF()
    Creates a model of a HKEMF1 modulator
    """
    def HKEFM(self, un = [], mash = False):
        #number of samples
        ns = un.size
        #output array
        yn = np.zeros(ns).astype(np.int64)
        y = 0
        #auxiliary arrays for checking number of intermidiate bits
        vn = np.zeros(ns).astype(np.int64)
        #error array
        e = np.zeros(ns).astype(np.int64)
        #quantizer specs
        b = self.in_bits+1 #expected quantizer input word length
```

```
        s = b - self.qbits #right shift
        #compute output feedback factor
        a = self.get_a()
        #simulation over time
        for n in range(1, ns):
            u = un[n]
            #accumulator
            v = u + e[n-1] + a * y
            #quantization
            y = self.quantize(v, s)
            #compute error feedback
            e[n] = v - (y << s-1)
            #output
            yn[n] = y
            #auxiliary arrays for checking number of intermidiate bits
            vn[n] = v

        print("---> First order HKEFM")
        print("     Number of bits before truncation:", len(bin(max(abs(vn))))-1)
        print("     Number of bits of quantization error:",
len(bin(max(abs(e))))-1)
        print("     Left shifting by", s, "bits to feedback the output")
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("     Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        if mash:
            return yn, e
        return yn

    """
    EMF2()
    Creates a model of a EMF2 modulator
    """
    def EFM2(self, un = []):
        #number of samples
        ns = un.size
        #output array
        yn = np.zeros(ns).astype(np.int64)
        #auxiliary arrays for checking number of intermidiate bits
        vn = np.zeros(ns).astype(np.int64)
        #error array
        e = np.ones(ns).astype(np.int64)
        #quantizer specs
        b = self.in_bits+1 #expected quantizer input word length
        s = b - self.qbits #right shift
        #simulation over time
        for n in range(2, ns):
            #input summation
            v = int(un[n] + 2 * e[n-1] - e[n-2])
            #quantization
            y = self.quantize(v, s)
            #compute error feedback
            e[n] = v - (y << s-1)
            #output
            yn[n] = y
            #auxiliary arrays for checking number of intermidiate bits
            vn[n] = v

        print("---> Second order EFM")
        print("     Number of bits before truncation:", len(bin(max(abs(vn))))-1)
```

```
        print("     Number of bits of quantization error:",
len(bin(max(abs(e))))-1)
        print("     Left shifting by", s, "bits to feedback the output")
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("     Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        return yn

    """
    CIFB()
    Creates a model of a CIFB modulator
    """
    def CIFB(self, un):
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
            #create NTF for CIFB
            NTF = ds.synthesizeNTF(order = self.order, osr = self.osr,
                                   opt=1, H_inf = self.H_inf)
            form = 'CIFB'
            a, g, b, c = ds.realizeNTF(NTF, form)
            ABCD = ds.stuffABCD(a, g, b, c, form)
            #scale coefficients for right output value
            ABCDscaled_out = ds.scaleABCD(ABCD, nlev = 2**(self.qbits))
            ABCDscaled = ABCDscaled_out[0]
            umax = ABCDscaled_out[1]
            a, g, b, c = ds.mapABCD(ABCDscaled, form)
            #coefficients rounding to shifts
            A, B, C, G, a, b, c, g = self.coef2shifts(a, b, c, g)
            #compute new NTF after rounding
            ABCDs = ds.stuffABCD(a, g, b, c, form)
            NTF, STF = ds.calculateTF(ABCDs)
        #number of samples
        ns = un.size
        #output array
        yn = np.zeros(ns).astype(np.int64)
        #intermidiate states array
        xx = [0 for i in range(self.order)]
        #auxiliary arrays for checking number of intermidiate bits
        xxx = [np.zeros(ns).astype(np.int64) for i in range(self.order)]
        vn = np.zeros(ns).astype(np.int64)
        #quantizer specs
        b = self.in_bits #expected quantizer input word length
        s = b - self.qbits #right shift
        #simulation over time
        for n in range(0, ns):
            u = int(un[n])
            #output summation
            vA = sum([c[0] * (xx[-1] >> abs(c[1])) if c[1] < 0 else
                      c[0] * (xx[-1] << c[1]) for c in C[-1]])
            vB = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                      b[0] * (u << b[1]) for b in B[-1]])
            v = vA + vB
            #quantization
            y = self.quantize(v, s)
            #output
            yn[n] = y
            #output feedback
            yf = y << s
            #intermidiate summations
            for j in range(self.order - 1):
```

```
                q = 2 + j
                x2A = sum([c[0] * (xx[-q] >> abs(c[1])) if c[1] < 0 else
                           c[0] * (xx[-q] << c[1]) for c in C[-q]])
                x2B = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                           b[0] * (u << b[1]) for b in B[-q]])
                x2C = sum([a[0] * (yf >> abs(a[1])) if a[1] < 0 else
                           a[0] * (yf << a[1]) for a in A[-(q-1)]])
                xx[-(q-1)] += x2A + x2B - x2C
            #input summation
            x1A = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                       b[0] * (u << b[1]) for b in B[0]])
            x1B = sum([a[0] * (yf >> abs(a[1])) if a[1] < 0 else
                       a[0] * (yf << a[1]) for a in A[0]])
            xx[0] += x1A - x1B
            xx[-2] -= sum([g[0] * (xx[-1] >> abs(g[1])) if g[1] < 0 else
                           g[0] * (xx[-1] << g[1]) for g in G[0]])

            #auxiliary arrays for checking number of intermidiate bits
            for p in range(len(xxx)):
                xxx[p][n] = xx[p]
            vn[n] = v

        print("---> Order", self.order, "CIFB")
        print("     Number of bits before truncation:", len(bin(max(abs(vn))))-1)
        for p in range(len(xxx)):
            print("     Number of bits before integrator", p+1, ":",
len(bin(max(abs(xxx[p])))))-1)
        print("     Left shifting by", s+1, "bits to feedback the output")
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("     Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        return yn

    """
    CRFB()
    Creates a model of a CRFB modulator
    """
    def CRFB(self, un):
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
            #create NTF for CRFB
            NTF = ds.synthesizeNTF(order = self.order, osr = self.osr, opt=1,
                                   H_inf = self.H_inf)
            form = 'CRFB'
            a, g, b, c = ds.realizeNTF(NTF, form)
            ABCD = ds.stuffABCD(a, g, b, c, form)
            #scale coefficients for right output value
            ABCDscaled_out = ds.scaleABCD(ABCD, nlev = 2**(self.qbits))
            ABCDscaled = ABCDscaled_out[0]
            umax = ABCDscaled_out[1]
            a, g, b, c = ds.mapABCD(ABCDscaled, form)
            #coefficients rounding to shifts
            A, B, C, G, a, b, c, g = self.coef2shifts(a, b, c, g)
            #compute new NTF after rounding
            ABCDs = ds.stuffABCD(a, g, b, c, form)
            NTF, STF = ds.calculateTF(ABCDs)
        #number of samples
        ns = un.size
        #output array
        yn = np.zeros(ns).astype(np.int64)
```

```
        #intermidiate states array
        xx = [0 for i in range(self.order)]
        #auxiliary arrays for checking number of intermidiate bits
        xxx = [np.zeros(ns).astype(np.int64) for i in range(self.order)]
        vn = np.zeros(ns).astype(np.int64)
        #quantizer specs
        b = self.in_bits #expected quantizer input word length
        s = b - self.qbits #right shift
        #simulation over time
        for n in range(0, ns):
            u = int(un[n])
            #output summation
            vA = sum([c[0] * (xx[-1] >> abs(c[1])) if c[1] < 0 else
                      c[0] * (xx[-1] << c[1]) for c in C[-1]])
            vB = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                      b[0] * (u << b[1]) for b in B[-1]])
            v = vA + vB
            #quantization
            y = self.quantize(v, s)
            #output
            yn[n] = y
            #output feedback
            yf = y << s
            #input summation
            x1A = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                       b[0] * (u << b[1]) for b in B[0]])
            x1B = sum([a[0] * (yf >> abs(a[1])) if a[1] < 0 else
                       a[0] * (yf << a[1]) for a in A[0]])
            xx[0] += x1A - x1B
            xx[-2] -= sum([g[0] * (xx[-1] >> abs(g[1])) if g[1] < 0 else
                           g[0] * (xx[-1] << g[1]) for g in G[0]])
            #intermidiate summations
            for j in range(self.order - 1):
                q = 2 + j
                x2A = sum([c[0] * (xx[-q] >> abs(c[1])) if c[1] < 0 else
                           c[0] * (xx[-q] << c[1]) for c in C[-q]])
                x2B = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                           b[0] * (u << b[1]) for b in B[-q]])
                x2C = sum([a[0] * (yf >> abs(a[1])) if a[1] < 0 else
                           a[0] * (yf << a[1]) for a in A[-(q-1)]])
                xx[-(q-1)] += x2A + x2B - x2C
            #auxiliary arrays for checking number of intermidiate bits
            for p in range(len(xxx)):
                xxx[p][n] = xx[p]
            vn[n] = v

        print("---> Order", self.order, "CRFB")
        print("     Number of bits before truncation:", len(bin(max(abs(vn))))-1)
        for p in range(len(xxx)):
            print("     Number of bits before integrator", p+1, ":",
len(bin(max(abs(xxx[p]))))-1)
        print("     Left shifting by", s+1, "bits to feedback the output")
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("     Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        return yn
    """
    CIFF()
    Creates a model of a CIFF modulator
    """
```

```
    def CIFF(self, un):
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
            #create NTF for CIFF
            NTF = ds.synthesizeNTF(order = self.order, osr = self.osr, opt=1,
                                   H_inf = self.H_inf)
            form = 'CIFF'
            a, g, b, c = ds.realizeNTF(NTF, form)
            ABCD = ds.stuffABCD(a, g, b, c, form)
            #scale coefficients for right output value
            ABCDscaled_out = ds.scaleABCD(ABCD, nlev = 2**(self.qbits))
            ABCDscaled = ABCDscaled_out[0]
            umax = ABCDscaled_out[1]
            a, g, b, c = ds.mapABCD(ABCDscaled, form)
            #coefficients rounding to shifts
            A, B, C, G, a, b, c, g = self.coef2shifts(a, b, c, g)
            #compute new NTF after rounding
            ABCDs = ds.stuffABCD(a, g, b, c, form)
            NTF, STF = ds.calculateTF(ABCDs)
        #number of samples
        ns = un.size
        #output array
        yn = np.zeros(ns).astype(np.int64)
        #intermidiate states array
        xx = np.ones(self.order).astype(np.int64())
        #auxiliary arrays for checking number of intermidiate bits
        xxx = [np.zeros(ns).astype(np.int64) for i in range(self.order)]
        vn = np.zeros(ns).astype(np.int64)
        #quantizer specs
        b = self.in_bits #expected quantizer input word length
        s = b - self.qbits #right shift
        #simulation over time
        for n in range(0, ns):
            u = int(un[n])
            #output summation
            vA = 0
            for i, x in enumerate(xx):
                vA += sum([a[0] * (x >> abs(a[1])) if a[1] < 0 else
                           a[0] * (x << a[1]) for a in A[i]])
            vB = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                      b[0] * (u << b[1]) for b in B[-1]])
            v = vA + vB
            #quantization
            y = self.quantize(v, s)
            #output
            yn[n] = y
            #output feedback
            yf = y << s
            #intermidiate summations
            for j in range(self.order - 1):
                q = 2 + j
                x2A = sum([c[0] * (xx[-q] >> abs(c[1])) if c[1] < 0 else
                           c[0] * (xx[-q] << c[1]) for c in C[-q+1]])
                x2B = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                           b[0] * (u << b[1]) for b in B[-q]])
                xx[-(q-1)] += x2A + x2B
            #input summation
            x1A = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                       b[0] * (u << b[1]) for b in B[0]])
            x1B = sum([c[0] * (yf >> abs(c[1])) if c[1] < 0 else
                       c[0] * (yf << c[1]) for c in C[0]])
            xx[0] += x1A - x1B
            xx[-2] -= sum([g[0] * (xx[-1] >> abs(g[1])) if g[1] < 0 else
                           g[0] * (xx[-1] << g[1]) for g in G[0]])
```

```
            #auxiliary arrays for checking number of intermidiate bits
            for p in range(len(xxx)):
                xxx[p][n] = xx[p]
            vn[n] = v

        print("---> Order", self.order, "CIFF")
        print("     Number of bits before truncation:", len(bin(max(abs(vn))))-1)
        for p in range(len(xxx)):
            print("     Number of bits before integrator", p+1, ":",
len(bin(max(abs(xxx[p])))))-1)
        print("     Left shifting by", s, "bits to feedback the output")
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("     Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        return yn

    """
    CRFF()
    Creates a model of a CRFF modulator
    """
    def CRFF(self, un):
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
            #create NTF for CRFF
            NTF = ds.synthesizeNTF(order = self.order, osr = self.osr, opt=1,
                                   H_inf = self.H_inf)
            form = 'CRFF'
            a, g, b, c = ds.realizeNTF(NTF, form)
            ABCD = ds.stuffABCD(a, g, b, c, form)
            #scale coefficients for right output value
            ABCDscaled_out = ds.scaleABCD(ABCD, nlev = 2**(self.qbits))
            ABCDscaled = ABCDscaled_out[0]
            umax = ABCDscaled_out[1]
            a, g, b, c = ds.mapABCD(ABCDscaled, form)
            #coefficients rounding to shifts
            A, B, C, G, a, b, c, g = self.coef2shifts(a, b, c, g)
            #compute new NTF after rounding
            ABCDs = ds.stuffABCD(a, g, b, c, form)
            NTF, STF = ds.calculateTF(ABCDs)
        #number of samples
        ns = un.size
        #output array
        yn = np.zeros(ns).astype(np.int64)
        #intermidiate states array
        xx = np.ones(self.order).astype(np.int64())
        #auxiliary arrays for checking number of intermidiate bits
        xxx = [np.zeros(ns).astype(np.int64) for i in range(self.order)]
        vn = np.zeros(ns).astype(np.int64)
        #quantizer specs
        b = self.in_bits #expected quantizer input word length
        s = b - self.qbits #right shift
        #simulation over time
        for n in range(0, ns):
            u = int(un[n])
            #output summation
            vA = 0
            for i, x in enumerate(xx):
                vA += sum([a[0] * (x >> abs(a[1])) if a[1] < 0 else
                           a[0] * (x << a[1]) for a in A[i]])
            vB = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
```

```
                    b[0] * (u << b[1]) for b in B[-1]])
            v = vA + vB
            #quantization
            y = self.quantize(v, s)
            #output
            yn[n] = y
            #output feedback
            yf = y << s
            #input summation
            x1A = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                       b[0] * (u << b[1]) for b in B[0]])
            x1B = sum([c[0] * (yf >> abs(c[1])) if c[1] < 0 else
                       c[0] * (yf << c[1]) for c in C[0]])
            xx[0] += x1A - x1B
            xx[-2] -= sum([g[0] * (xx[-1] >> abs(g[1])) if g[1] < 0 else
                           g[0] * (xx[-1] << g[1]) for g in G[0]])
            #intermidiate summations
            for j in range(self.order - 1):
                q = 2 + j
                x2A = sum([c[0] * (xx[-q] >> abs(c[1])) if c[1] < 0 else
                           c[0] * (xx[-q] << c[1]) for c in C[-q+1]])
                x2B = sum([b[0] * (u >> abs(b[1])) if b[1] < 0 else
                           b[0] * (u << b[1]) for b in B[-q]])
                xx[-(q-1)] += x2A + x2B
            #auxiliary arrays for checking number of intermidiate bits
            for p in range(len(xxx)):
                xxx[p][n] = xx[p]
            vn[n] = v

        print("---> Order", self.order, "CRFF")
        print("     Number of bits before truncation:", len(bin(max(abs(vn))))-1)
        for p in range(len(xxx)):
            print("     Number of bits before integrator", p+1, ":",
len(bin(max(abs(xxx[p]))))-1)
        print("     Left shifting by", s, "bits to feedback the output")
        if self.qtype == 1:
            M = max(abs(yn))+1
            outbits = np.log2(M)
        else:
            M = 2*max(abs(yn))+1
            outbits = np.log2(M)
        print("     Word length at output:", outbits, "~", int(np.ceil(outbits)),
"bits for", M, "levels")
        return yn
```

A4. Auxiliary Python Code

```
import deltasigma as ds
import numpy as np
from tqdm import tqdm
import time as time_module
import scipy.signal as sig

def SFDR_THD(yff, fin, f, fs, nhar, window_size = 0):
    fbin = np.where(f == fin)[0][0]
    Psignal = 10*np.log10(sum(10**(yff[fbin-window_size:fbin+1+window_size]/10)))
    hars_nums = np.arange(2, nhar+2, 1)
    fhars =  hars_nums * fin
    k1 = np.zeros(1)
    k2 = np.zeros(1)
    while k1.size or k2.size:
```

```
        k1 = np.where(fhars > f.max()/2)[0]
        fhars[k1] = np.abs(fs - fhars[k1])
        k2 = np.where(fhars > f.max())[0]
        fhars[k2] = fhars[k2] - fs
    har_bins = []
    for fhar in fhars:
        har_bins.append(np.where(f == fhar)[0][0])
    max_har_bin = har_bins[np.where(yff[har_bins] == yff[har_bins].max())[0][0]]
    Phar2 = 10*np.log10((10**(yff[max_har_bin-window_size:max_har_bin+1+win-
dow_size]/10)).sum())
    SFDR = Psignal - Phar2
    extended_har_bins = []
    for i in range(-window_size, window_size+1):
        extended_har_bins += (np.array(har_bins)+i).tolist()
    har_sum = (10**(yff[extended_har_bins]/10)).sum()
    Phar = 10*np.log10(har_sum)
    THD = Phar - Psignal
    return SFDR, THD, har_bins, max_har_bin

def computeFFT(y, fin, bw, fs, ns, dbc = False, snr = False,
               phase = False, window_order = 1, Complex = False):
    w = ds.ds_hann(ns)**window_order
    fy = np.fft.fft(y * w)/(np.linalg.norm(w, 1)/2)
    fy_power = abs(fy)**2
    fy_power_dB = ds.dbp(fy_power)
    if dbc:
        fy_power_dB -= fy_power_dB[int(fin/(fs) * ns)]
    if snr:
        window_order += 3
        bw_bin = int(bw/(fs) * ns)
        fbin = int(fin/(fs) * ns)
        snr_calc = ds.calculateSNR(fy[2+window_order:bw_bin],
                                   fbin-2-window_order, nsig=window_order)
        #alternatively
        # PS = sum(abs(fy[fbin-window_order:fbin+window_order+1])**2)
        # PN = sum(abs(fy[2+window_order:fbin-window_order])**2) +
sum(abs(fy[fbin+window_order+1:bw_bin])**2)
        # snr_calc = 10*np.log10(PS/PN)

    if phase:
        fy_phase = np.arctan(np.imag(fy)/np.real(fy))
    if phase and snr:
        return fy_power_dB, snr_calc, fy_phase
    if phase and not snr:
        return fy_power_dB, fy_phase
    if not phase and snr:
        return fy_power_dB, snr_calc
    if Complex:
        return fy_power_dB, fy
    return fy_power_dB

def shunt(r1, r2):
    return (r1*r2)/(r1+r2)

class Constants:
    def __init__(self, T):
        self.gama = 1
        self.K = 1.380649e-23
        self.T = T

def computePinkNoise(bw0, fc, kf, gm, time):
    print("Computing flicker noise...")
    nsf = int(500 * np.log10(fc/bw0))
    f =  np.linspace(bw0, fc, nsf)
    # f = np.logspace(np.log10(bw0), np.log10(fc), nsf)
```

```
    deltaf = (fc-bw0)/nsf
    noise = np.zeros(time.size)
    for fi in tqdm(f):
        amp = gm * np.sqrt(2*kf*np.log(1+deltaf/fi) - 2*kf*np.log(1+deltaf/fc))
        noise += amp*np.sin(2*np.pi*fi * time + np.random.uniform()*2*np.pi)
    print("")
    return noise

def maxminNL(NL):
    if abs(max(NL)) > abs(min(NL)):
        return max(NL)
    else:
        return min(NL)

def smooth_spectrum(f, fft, fft_no_spurs, fin):
    window = 2**14
    cumsum_vec = np.cumsum(np.insert(fft_no_spurs, 0, 0))
    smooth_fft = np.zeros(f.size)
    # Suppress/hide the warning
    np.seterr(invalid='ignore')
    smooth_fft[2*window-1:] = (cumsum_vec[window:][window:] - cumsum_vec[win-
dow:][:-window]) / window
    last_window = window
    while window >= 2:
        window = int(window/2)
        smooth_fft[2*window:2*last_window] = (cumsum_vec[window:2*last_win-
dow][window:] - cumsum_vec[window:2*last_window][:-window]) / window
        last_window = window
    smooth_fft[0:2*window] = fft_no_spurs[0:2*window]
    smooth_fft[0:3] = smooth_fft[3]
    spurs = np.where(f % fin == 0)[0][1:11]
    for spur in spurs:
        if fft[spur] > smooth_fft[spur]:
            smooth_fft[spur] = fft[spur]
    return smooth_fft

def spurs_removal(fin, f, fft, window_size):
    new_fft = fft.copy()
    spurs = np.where(f[:-window_size] % fin == 0)[0][:-1]
    for spur in spurs:
        new_fft[spur] = (fft[spur+1+window_size] + fft[spur-1-window_size])/2
        for i in range(1, window_size+1):
            new_fft[spur+i] = fft[spur+1+window_size]
            new_fft[spur-i] = fft[spur-1-window_size]
    return new_fft

def is_prime(n):
    """
    Assumes that n is a positive natural number
    """
    # We know 1 is not a prime number
    if n == 1:
        return False
    i = 2
    # This will loop from 2 to int(sqrt(x))
    while i*i <= n:
        # Check if i divides x without leaving a remainder
        if n % i == 0:
            # This means that n has a factor in between 2 and sqrt(n)
            # So it is not a prime number
            return False
        i += 1
    # If we did not find any factor in the above loop,
    # then n is a prime number
    return True
```

A5. CSDAC Python Code

```
import numpy as np
from bitstring import Bits
import aux_sdm as aux
from tqdm import tqdm
from numpy.random import normal
from scipy.signal import filtfilt, butter
import matplotlib.pyplot as plt
import pandas as pd

def diff_CSDAC(un, nbits, seg, CL, RL, VFS, omega, Rout, fs, time, gm,
               fn0, fnc, nppclk = 1, test = False, DAC_noise = False):
    #binary weighted bits
    bwbits = int((1-seg)*nbits)
    #unary weighted bits
    tcbits = int(np.ceil(seg*nbits))
    #number of levels in the converter
    N = 2**nbits
    #LSB current value
    Iref = VFS/(2*RL*(N-1))

    print("\n___Differential Current Steering DAC___\n")
    print("Resolution:", nbits)
    print("Segmentation level:", seg, "-->", bwbits,
          "binary weighted bits and", tcbits, "unary weighted bits")
    print("LSB current:", Iref, "A")
    print("LSB relative current standard deviation σ/I:", omega)
    print("LSB current source output impedance:", Rout, "Ω")
    print("")

    #get Zout characteristic from csv file
    try:
        nofile = False
        Rout_array, V_array = Zout('cadence\\Zout_DC11.csv')
        Rout_array = Rout_array/np.abs(Rout_array).min() * Rout
    except FileNotFoundError:
        RoutP = Rout
        RoutN = Rout
        nofile = True

    #noise corner frequency and flicker noise upper bound
    _4KT = 4*1.380649e-23*300
    fnc = 1e5
    #compute flicker noise coefficient
    kf =  _4KT/gm * fnc
    #flicker noise lower bound
    fn0 = 1e1

    #Signal characteristics
    if test:
        nppclk = 1
        un = np.arange(0, 2**(nbits), 1)
        time = np.arange(0, 2**nbits/fs, 1/fs)
    ns = un.size
    y = np.zeros(ns * nppclk)
    yf = np.zeros(ns * nppclk)
    VoutP = np.zeros(ns)
    VoutN = np.zeros(ns)
    IPbw = 0
    INbw = 0
    IPtc = 0
    INtc = 0
```

```
    #Create termometer code current sources
    tc_array = np.array([np.sum(np.array([Iref * (1 + normal(0, omega))
                                          for j in range(2**bwbits)]))
                         for i in range(int(2**tcbits-1))])
    tc_arrayP = np.insert(np.cumsum(tc_array), 0, 0)
    tc_arrayN = np.insert(np.cumsum(np.flip(tc_array)), 0, 0)
    #Create binary weighted current sources
    bw_array = np.flip(np.array([sum(np.array([Iref * (1 + normal(0, omega))
                                               for i in range(int(2**j))]))
                                 for j in range(bwbits)]))
    #Compute flicker noise array if wanted
    if DAC_noise:
        fnoise_array = aux.computePinkNoise(fn0, fnc, kf, gm, time)
    #Simulate DAC
    print("Time domain simulation...")
    for n, u in enumerate(tqdm(un)):
        u_ = 2**nbits-u-1
        #computing binary weighted current
        if bwbits != 0:
            bbw = (Bits(uint = u, length = nbits).bin)[tcbits:]
            ubw = int(bbw, 2)
            bintbw = np.array([int(b) for b in bbw])
            IPbw = bintbw.dot(bw_array)
            not_bintbw = np.array([int(not b) for b in bintbw])
            INbw = not_bintbw.dot(bw_array)
        else:
            ubw = IPbw = 0
        ubw_ = 2**bwbits - 1 - ubw
        #computing thermometer coded current
        utc = (u >> bwbits)
        IPtc = tc_arrayP[utc]
        utc_ = len(tc_arrayP) - utc - 1
        INtc = tc_arrayN[utc_]

        #computing output resistance
        if not nofile:
            if n == 0:
                RoutP = Rout_array[0]
                RoutN = Rout_array[-1]
            else:
                RoutP = Rout_array[np.abs((V_array - VoutP[n-1])).argmin()]
                RoutN = Rout_array[np.abs((V_array - VoutN[n-1])).argmin()]
        #RL in parallel with u or u_ current source output impedances
        if u != 0:
            ReqP = abs(aux.shunt(RL, RoutP/(u)))
        else:
            ReqP = RL
        if u_ != 0:
            ReqN = abs(aux.shunt(RL, RoutN/(u_)))
        else:
            ReqN = RL
        #compute DAC thermal and flicker noise if wanted
        if DAC_noise != 0:
            #current source thermal noise
            tnoise = np.random.normal(0, np.sqrt(
                2 * ns * _4KT * ((N-1) * gm * RL**2 + 2 * RL)))
            #current source flicker noise
            fnoise = fnoise_array[n] * np.sqrt(N-1) * RL
            #total output noise
            noise = tnoise + fnoise
        else:
            noise = 0
        #linear output current
        IoutP = IPtc + IPbw
        IoutN = INtc + INbw
```

```
        #computing output voltage
        y[n * nppclk: (n+1) * nppclk] += IoutP * ReqP - IoutN * ReqN + noise
        #computing output test voltage
        VoutP[n] = IoutP * ReqP
        VoutN[n] = IoutN * ReqN
    #add settling time to output voltage by RC circuit
    tau = RL*CL
    yf = y * (1-np.exp(-time/tau))
    print("")
    return yf

def testDAC(un, y, nbits):
    VLSB = (y[-1]-y[0])/(2**nbits-1)
    yideal = np.linspace(y[0], y[-1], 2**nbits)
    INL = np.zeros(y.size)
    DNL = np.zeros(y.size-1)
    for i, v in enumerate(y):
        INL[i] = (v - yideal[i])/VLSB
        if i < DNL.size:
            DNL[i] = (y[i+1] - v - VLSB)/VLSB
    gain = (y[-1] - y[0])/((un[-1] - un[0]) * VLSB)
    return VLSB, yideal, INL, DNL, gain


def omega2_I2_from_INL(INL_yield, INLmax, M):
    from scipy.stats import norm
    import numpy as np
    omega2_I2 = INLmax/(norm.ppf(0.5 + INL_yield/2) * np.sqrt(2**(M)))
    return omega2_I2

def rout_for_SFDR(M, RL, SFDR):
    return 0.25*RL*(2**M)*10**(SFDR/40)

def Zout(file):
    df = pd.read_csv(file)
    cols = list(df.columns)
    df.rename(columns = {cols[0]: "output voltage [V]", cols[1]: "Re(Zout)
[Ohm]", cols[2]: "Im(Zout) [Ohm]"}, inplace = True)
    Rout_array = -np.array(df["Re(Zout) [Ohm]"].tolist()) - 1j * np.ar-
ray(df["Im(Zout) [Ohm]"].tolist())
    V_array = np.array(df["output voltage [V]"].tolist())
    return Rout_array, V_array
```

A6. gm-C Filter Python Code

```
import scipy.signal as sig
import numpy as np
import aux_sdm as aux

def rec_filt(x, N, wo, Q, gm, fs, go, Cp, tau, time, fn0, fnc, filter_noise =
False):
    print("\n___gm-C Low Pass Filter___\n")
    print("Filter order:", N)
    print("Cutoff frequency:", 0.5*wo/np.pi, "Hz")

    #number of samples
    ns = x.size
    _4KT = 4*1.380649e-23*300
    fnoise1 = np.zeros(ns)
    fnoise2 = np.zeros(ns)
    noise = np.zeros(ns)
    print("gm = {:e} S".format(gm))
```

```
    #first order filter for odd order filter implementations
    if N%2 != 0:
        #filter transfer function and frequency response
        C1 = gm/wo
        print("C1 >= {:e} F".format(C1))
        Cp1 = np.random.normal(0, Cp/3) * C1
        go1 = np.random.normal(0, go/3) * gm
        num1 = [gm * tau, -gm]
        den1 = [C1 + Cp1 - gm*tau, gm + go1]
        numz1, denz1 = sig.bilinear(num1, den1, fs)
        f1, h1 = sig.freqz(numz1, denz1, worN = np.arange(0, fs, fs/ns), fs = fs)
        h = h1
        f = f1
        #filter noise
        if filter_noise:
            noise1 = _4KT/gm
            max_noise = noise1
            kf1 =  _4KT/gm * fnc
            fnoise1 = aux.computePinkNoise(fn0, fnc, kf1, gm, time)
            noise = np.random.normal(0, np.sqrt(2 * ns * max_noise), ns)
            noise += fnoise1/gm
    #second order filter for orders bigger than 1
    if N != 1:
        #filter transfer function and frequency response
        C22 = gm/(wo*Q)
        C21 = C22*Q**2
        D = gm*tau
        go2 = np.random.normal(0, go/3)*gm
        C21 *= (1+np.random.normal(0, Cp/3))
        C22 *= (1+np.random.normal(0, Cp/3))
        print("C21 = {:e} F".format(C21))
        print("C22 >= {:e} F".format(C22))
        a2 = D**2/(D**2-C22*D+C21*C22)
        a1 = -gm*2*D/(D**2-C22*D+C21*C22)
        a0 = gm**2/(D**2-C22*D+C21*C22)
        b2 = 1
        b1 = ((3*C22+C21-D)*go2 + gm*(C22-2*D))/(D**2-C22*D+C21*C22)
        b0 = (gm**2 + 3*go2**2 + gm*go2)/(D**2-C22*D+C21*C22)
        num2 = [a2, a1, a0]
        den2 = [b2, b1, b0]
        numz2, denz2 = sig.bilinear(num2, den2, fs)
        f2, h2 = sig.freqz(numz2, denz2, worN = np.arange(0, fs, fs/ns), fs = fs)
        f = f2
        #filter noise
        if filter_noise:
            noise2 = 2 *_4KT/gm
            max_noise = noise2 * int(N/2)
            kf2 =  _4KT/gm * fnc
            fnoise2 = aux.computePinkNoise(fn0, fnc, kf2, gm, time)
    #time domain filtering and noise
    if N%2 == 0:
        #second order cascade
        h = h2**(N/2)
        if filter_noise:
            noise = np.random.normal(0, np.sqrt(2 * ns * max_noise), ns)
            noise += (fnoise2/gm)*(N/2)
        x += noise
        y = sig.lfilter(numz2, denz2, x)
        for n in range(int(N/2)-1):
            y = sig.lfilter(numz2, denz2, y)
    else:
        #odd order
        if N != 1:
            h *= h2**(int(N/2))
            if filter_noise:
```

```
                noise += (fnoise2/gm) * int(N/2)
        x += noise
        y = sig.lfilter(numz1, denz1, x)
        for n in range(int(N/2)):
            y = sig.lfilter(numz2, denz2, y)
    return f, h, y, noise
```

A7. Main Simulation Python Code

```
import deltasigma as ds
import numpy as np
import matplotlib.pyplot as plt
import dsdm as modulator
import aux_sdm as aux
import scipy.signal as sig
import rec_filt as filt
import matplotlib
import dac

font = {'family' : 'DejaVu Sans',
        'weight' : 'normal',
        'size'   : 12}

matplotlib.rc('font', **font)
%matplotlib qt

#--------------------system and simulation variables---------------------
#input word length
in_bits = 48
#sampling frequency (Hz)
fs = 122.88e3
#number of samples
ns = 2**17
#number of periods
k = 7
periods = ns/(2**k) + 1
#input frequency (Hz)
fin = fs * periods/ns
#relative input amplitude
Ain = 0.7
#bandwidth for sndr calculation (Hz) (brick wall filter assumption w/ cutoff @
bw)
bw = 1e3
#oversamplig ratio
osr = fs/(2*fin)
#sine period
T = 1/fin #[s]
#sampling period
dt = 1/fs #[s]
#sine frequency bin
fbin = int((fin/(fs)) * ns)
#fft bin width
deltaf = fs/ns
#frequencies array
f = np.arange(0, fs, deltaf)
#half frequencies array
f_2 = int(f.size/2)

print("\n___System Simulation___\n")
print("Input word length:", in_bits, "bits signed")
print("Sampling frequency:", fs, "Hz")
print("Oversampling ratio:", osr)
```

```
print("Number of samples:", ns)
print("Relative input amplitude:", Ain*100, "%")
print("Bandwidth for SNDR calculation:", bw, "Hz")

#------------------------------noise specs------------------------------
#phase noise specs after VCXO
pn = np.array([-100, -110, -115, -130, -138,
               -145, -153, -161, -165, -166, -166])
#offset frequencies
fpn = np.array([3e1, 7e1, 1e2, 3e2, 1e3, 3e3, 1e4, 1e5, 1e6, 1e7, 5e7])
#VCXO gain
Kv = 25e-6 * fs
#VCXO output amplitude [V]
Amp = 1.65
#SFDR spec after VCXO
SFDR = 100
#jitter calculation
integ = 0
for i in range(len(fpn)-1):
    integ += (fpn[i+1]-fpn[i]) * (10**(pn[i+1]/10)
                                  + 0.5 * (10**(pn[i]/10)-10**(pn[i+1]/10)))
jitter = np.sqrt(2*integ)/(2*np.pi*fs)
#VCXO input noise spec
noise = ds.dbp((10**(pn/10)*fpn**2*Amp**2/Kv**2))
#VCXO input SFDR spec
sfdr = ds.dbp((10**(-SFDR/10)*2**2*fpn**2*Amp**2/Kv**2))

#-------------Digital ΣΔ Modulator variables (refer to dsdm.py)---------------
#Modulator NTF high frequency gain
H_inf = 1.5
#Modulator [*100 %] maximum rounding to power of two coefficient error
max_error = 0.01
#Modulator gain
gain = 1
#Modulator type
qtype = 1
#Modulator topology
top = "EFM"
#Modulator order
order = 2
#Modulator quantization bits
qbits = 1
#window order (for FFT)
window_order = order + 1

#-------------Current Steering DAC variables (refer to dac.py)---------------
#DAC relative current standard deviation
omega = 0.001
#DAC current source output impedance (Ω)
Rout = 1e6
#DAC current source gm (S)
gm = 100e-6
#Resistive load (Ω)
RL = 50
#Capacitive load (F)
CL = 10e-12
#Segmentation level -> (thermometer bits - binary bits) / total number of bits
seg = 1
#Full-scale differential voltage (V)
VFS = 1
#Consider thermal and flicker noise on DAC
DAC_noise = True
#flicker noise lower bound (Hz)
fn0_DAC = 1e1
#flicker noise corner frequency (Hz)
```

```
fnc_DAC = 1e3

#-------------gm-C filter variables (refer to rec_filt.py)---------------
#filter order
filter_order = 5
#filter cutoff frequency
cutoff_freq = 5e3
#filter quality factor
Qfactor = 1/np.sqrt(2)
#filter transconductance (S)
filter_gm = 80e-6
#Consider thermal and flicker noise on filter
filter_noise = True
#flicker noise lower bound (Hz)
fn0_filter = 1e1
#flicker noise corner frequency (Hz)
fnc_filter = 1e3

#-----------------------discrete time simulation-----------------------
#time array for simulation
t1 = np.arange(0, ns * dt, dt)
t1 += np.random.normal(0, jitter, t1.size) #add jitter

#Input digital signal
un = Ain * (2**(in_bits-1) - 1) * (np.sin(2 * np.pi * fin * t1))
un = np.round(un).astype(np.int64())

#Sigma-delta modulator
dsdm = modulator.DSDM(osr = osr, top = top, order = order,
                      qbits = qbits, in_bits = in_bits, qtype=qtype,
                      H_inf = H_inf, max_err = max_error, gain = gain)
yd = dsdm.mod(un)
ydd = yd-min(yd)

#Differential current steering DAC
yv = dac.diff_CSDAC(ydd, nbits=int(np.ceil(np.log2(max(ydd)+1))), seg=seg, CL=CL,
                    RL=RL, VFS = VFS, omega = omega,
                    Rout=Rout, fs=fs, time=t1, gm=gm,
                    fn0 = fn0_DAC, fnc = fnc_DAC, DAC_noise = DAC_noise)
yvv = yv.copy()

#gm-C filtering
ff, hf, yf, filter_noise = filt.rec_filt(yvv, N=filter_order, wo=2*np.pi*cut-
off_freq,
                                         Q=Qfactor, gm=filter_gm,
                                         fs=fs, go=0, Cp=0, tau=0, time=t1,
                                         fn0 = fn0_filter, fnc = fnc_filter,
                                         filter_noise = filter_noise)

#FFTs
ydf, snrd = aux.computeFFT(yd, fin, bw, fs, ns,
                      dbc = False, snr = True, window_order=window_order)

yvf, snrv = aux.computeFFT(yv, fin, bw, fs, ns,
                      dbc = False, snr = True, window_order=window_order)

yff, snrf = aux.computeFFT(yf, fin, bw, fs, ns,
                      dbc = False, snr = True, window_order=window_order)

yff_copy = yff.copy()
yff_copy -= 10*np.log10(deltaf)
yff_copy -= yff_copy[fbin]
yff_copy2 = yff_copy.copy()
no_spurs = aux.spurs_removal(fin, f, yff_copy, window_order)
smooth = aux.smooth_spectrum(f, yff_copy2, no_spurs, fin)
```

```
cycles = np.arange(0, len(t1))
plt.figure(0)
plt.grid(True, which='both')
plt.step(cycles , yd, "b-", linewidth = 0.5, label="modulator's output")
plt.plot(cycles , max(abs(yd))*un/max(un), "r-", label="normalized input")
plt.xlabel("Cycles")
plt.title("Modulator's output")
plt.yticks([-max(yd), 0, max(yd)])
plt.xlim([0, round(3*T/dt)]) # first 3 periods
plt.legend(loc='lower right', prop={'size': 10})

plt.figure(1)
plt.grid(True, which='both')
plt.step(t1, yv, "b-", linewidth = 0.5)
plt.xlabel("Time [s]")
plt.xlim([0, 3*T]) # first 3 periods
plt.title("DAC's output")

plt.figure(2)
plt.grid(True, which='both')
plt.plot(t1, yf, "b-", linewidth = 1)
plt.xlabel("Time [s]")
plt.xlim([0, 3*T]) # first 3 periods
plt.title("Filtered output")

plt.figure(3)
plt.grid(True, which='both')
plt.semilogx(f[2+window_order:f_2], ydf[2+window_order:f_2], "b-", linewidth =
0.5,
             label = top + "{:d}".format(order))
plt.xlabel("Frequency [Hz]")
plt.ylabel("[dB]")
plt.title("Modulator's output spectrum")

plt.figure(4)
plt.grid(True, which='both')
plt.semilogx(f[2+window_order:f_2], yvf[2+window_order:f_2], "b-", linewidth =
0.5,
             label = top + "{:d}".format(order))
plt.xlabel("Frequency [Hz]")
plt.ylabel("[dB]")
plt.title("DAC output spectrum")

plt.figure(5)
plt.grid(True, which='both')
plt.semilogx(f[2+window_order:f_2], yff[2+window_order:f_2], "b-", linewidth =
0.5,
             label = top + "{:d}".format(order))
plt.xlabel("Frequency [Hz]")
plt.ylabel("[dB]")
plt.title("Filter output spectrum")


plt.figure(6)
plt.grid(True, which="both")
plt.semilogx(fpn, noise, "--", marker = ".", color = "black",
              linewidth = 1, label = "Required input noise level")
plt.semilogx(fpn, sfdr, "--", color = "black",
              linewidth = 1, label = "Required SFDR level")
plt.semilogx(f[3:f_2], smooth[3:f_2], "b-", linewidth = 2,
              label = "Filtered " + top + "{:d}".format(order))
plt.xlabel("Frequency [Hz]")
plt.ylabel("[dBc]")
plt.legend(loc='upper left', prop={'size': 10})
```

```
plt.title("System output spectrum Vs required noise and SFDR performance @ VCXO
input")

plt.figure(7)
plt.grid(True, which="both")
noiseff = aux.computeFFT(filter_noise, 1, 1, 1, filter_noise.size,
                         dbc = False, snr = False,
                         phase = False, window_order = 0)
noisefff = noiseff - 10*np.log10(deltaf)
smooth_noisefff = aux.smooth_spectrum(ff, noisefff, noisefff, 0)
plt.semilogx(ff[1:f_2], noisefff[1:f_2], "b-", linewidth = 0.4)
plt.semilogx(ff[1:f_2], smooth_noisefff[1:f_2], "r--", linewidth = 3)
plt.xlabel("Frequency [Hz]")
plt.ylabel("[dB]")
plt.title("Filter noise spectrum")

plt.figure(8)
plt.grid(True, which="both")
plt.semilogx(ff[:f_2], 20*np.log10(abs(hf[:f_2])), "b-", linewidth = 1)
plt.xlabel("Frequency [Hz]")
plt.ylabel("[dB]")
plt.title("Filter frequency response")

SFDR, THD, har_bins, max_har_bin = aux.SFDR_THD(yff, fin, f, fs, nhar=10, win-
dow_size=window_order)

print("\n___Measurements___\n")
print("Modulator output signal power:", round(ydf[fbin], 2), "dB")
print("Modulator relative output signal amplitude:",
      round(10**((ydf[fbin]-6*(qbits-1))/20), 2), "with Ain =", Ain)
print("SNDR after ΣΔ Modulator:", round(snrd, 2), "dB")
print("SNDR after DAC:", round(snrv, 2), "dB")
print("SNDR after reconstruction filter:", round(snrf, 2), "dB")
print("SFDR after reconstruction filter:", round(SFDR, 2), "dBc")
print("THD after reconstruction filter:", round(THD, 2), "dBc")
for freq in fpn:
    freq2 = deltaf*int(freq/deltaf)
    freqbin = np.where(f[:f_2] == freq2)[0]
    if freqbin.size:
        phase_noise = smooth[freqbin][0] + 20*np.log10(Kv/(Amp*freq2))
        print("Phase noise @", freq, "Hz:", round(phase_noise, 2), "dBc/Hz")
```

A8. DAC Metrics from Electrical Simulation Python Code

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
import aux_sdm as aux
%matplotlib qt

font = {'family' : 'DejaVu Sans',
        'weight' : 'normal',
        'size'   : 16}

matplotlib.rc('font', **font)

nbits = 10
Fclk = 122.88e6
Tclk = 1/Fclk
VFS = 0.5
```

```
delay = 2e-6
clk_delay = 1e-9

df =  pd.read_csv('cadence\\TF.csv')
cols = list(df.columns)
df.rename(columns = {cols[0]: "time [s]", cols[1]: "output voltage [V]"},
          inplace = True)
t = np.array(df['time [s]'].tolist())
y = np.array(df["output voltage [V]"].tolist())
start = np.where(t >= delay + Tclk)[0][0]
t = t[start:]
y = y[start:]
tsteps = np.array([t.min() + Tclk*i + 0.25*Tclk for i in range(2**nbits)])
step_points = np.zeros(2**nbits).astype(int)
for i in range(2**nbits):
    step_points[i] = np.abs(tsteps[i]-t).argmin()
yideal = np.linspace(y[step_points[0]], y[step_points[-1]], 2**nbits)
VLSB = (y[step_points[-1]]-y[step_points[0]])/(2**nbits-1)
plt.figure(0)
plt.grid(which = 'both')
plt.plot(t, y)
plt.title("DAC Code-Voltage Transfer Function")
plt.xlabel("time [s]")
plt.ylabel("Output voltage [V]")
plt.plot(t[step_points], y[step_points], 'g*')

INL = (y[step_points] - yideal)/VLSB
DNL = (np.abs(y[step_points[1:]] - y[step_points[:-1]]) - VLSB)/VLSB

print("VLSB: {}".format(VLSB))

plt.figure(1)
plt.grid(which = 'both')
plt.plot(DNL, "b-", linewidth = 0.4, label = "DNL")
a = np.arange(1, 64, 1)*16 -1
plt.plot(a, DNL[a], "r+", label = "MSB step")
plt.ylabel("DNL [LSBs]")
plt.xlabel("Codes")
plt.title("DNL")
plt.legend(loc='best', prop={'size': 10})

plt.figure(2)
plt.grid(which = 'both')
# plt.plot(np.cumsum(DNL), "-", label = "cumsum(DNL)")
plt.plot(INL, "b-", linewidth = 0.4, label = "INL formula")
plt.ylabel("INL [LSBs]")
plt.xlabel("Codes")
# plt.legend()
plt.title("INL")

codes = np.arange(1, 2**nbits, 1)

plt.figure(0)
sim_time = t.max()-t.min()
tideal = t[step_points]
plt.step(tideal, yideal, "--k", where="mid")

DNLmax = max(abs(DNL))
INLmax = max(abs(INL))
print("DNL: " + str(round(DNLmax, 3)) + " LSB")
print("INL: " + str(round(INLmax, 3)) + " LSB")

number_of_corners = 1
for num in range(number_of_corners):
    corner = str(num)
```

```
    print("\ncorner: " + corner)
    df = pd.read_csv('cadence\\sine_60M_C' + corner + '+M.csv')
    cols = list(df.columns)
    df.rename(columns = {cols[0]: "time [s]", cols[1]: "output voltage [V]"},
              inplace = True)
    t = np.array(df['time [s]'].tolist())
    y = np.array(df["output voltage [V]"].tolist())

    fin = 60.21e6
    osr = 1
    fs = Fclk
    ns = 4096
    offset = y.size - ns
    window = 1

    yf, SNDR = aux.computeFFT(y[offset:ns+offset], fin, osr, fs,
                              ns, dbc=False, snr=True, window_order=window)
    deltaf = fs/ns
    f = np.arange(0, fs, deltaf)
    f_2 = int(f.size/2)
    nhar = 4
    SFDR, THD, harm_bins, sfdr_bin = aux.SFDR_THD(yf[:f_2], fin, f,
                                                  fs, nhar = nhar,
                                                  window_size=window)
    smooth_yf = aux.smooth_spectrum(f, yf)
    noise_level = smooth_yf.mean() - 10*np.log10(deltaf)
    fbin = int((fin/(fs)) * ns)

    plt.figure(3+num)
    plt.grid(True, which="both")
    plt.plot(f[:f_2], yf[:f_2], "-", linewidth = 0.4,
             label = "DAC output", color = "blue")
    plt.plot(f[harm_bins], yf[harm_bins], "+",
             label = "Harmonics", color = "red", markersize=10)
    plt.plot(f[sfdr_bin], yf[sfdr_bin], "x",
             label = "SFDR bin", color = "orange", markersize=10)
    for n in range(nhar):
        plt.text(f[harm_bins[n]], yf[harm_bins[n]] + 6,
                 "{:d}".format(n+2),
                 fontsize=12, horizontalalignment='center')
    plt.text(f[fbin]-3e6, -6,
             "{:.3f} dB".format(yf[fbin]), fontsize=9,
             horizontalalignment='center')
    plt.xlabel("Frequency [x10 MHz]")
    plt.ylabel("[dB]")
    plt.legend(loc='best', prop={'size': 10})
    plt.xlim(0, fs/2)
    plt.ylim(-120, 0)
    plt.title("C" + str(num))

    HD3 = yf[harm_bins[1]]
    maxSwing = max(abs(y))
    print("SNDR: " + str(round(SNDR, 2)) + " dB")
    print("ENOB: " + str(round((SNDR-1.76)/6.02, 2)) + " bits")
    print("SFDR: " + str(round(SFDR, 2)) + " dBc")
    print("THD: " + str(round(THD, 2)) + " dBc")
    print("HD3: " + str(round(HD3, 2)) + " dB")
    print("Max Swing: " + str(round(maxSwing*1000, 2)) + " mV")
```